

VIC-20

An all-purpose reference
guide for first-time
computerists as well
as experienced
programmers!

PROGRAMMER'S REFERENCE GUIDE



 **commodore**
COMPUTER

VM110

63 2023252
Clearance
Was
\$149.95 722

VIC 20 Programmer's Reference Guide

A. Finkel
N. Harris
P. Higginbottom
M. Tomczyk

Published by
Commodore Business Machines, Inc.
and Howard W. Sams & Co., Inc.

FIRST EDITION
Fourth printing—1982

Copyright © 1982 by Commodore Business Machines, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Commodore Business Machines, Inc.

TABLE OF CONTENTS

INTRODUCING THE PROGRAMMER'S REFERENCE GUIDE..... v

VIC 20 APPLICATIONS GUIDE..... vii

1 BASIC PROGRAMMING REFERENCE GUIDE..... 1

- VIC BASIC: The Language of the VIC..... 3
- Commands..... 5
- Statements..... 14
- I/O Statements..... 35
- BASIC Functions..... 40
- Numbers and Variables..... 54
- Operators..... 62
- Logical Operators..... 68

2 PROGRAMMING TIPS..... 71

- Editing Programs..... 73
- Using the GET Statement..... 77
- How to Crunch BASIC Programs..... 79
- Working With Graphics..... 82
 - Character Memory..... 82
 - Programmable Characters..... 82
 - High Resolution Graphics..... 88
 - Multi-Color Mode Graphics..... 92
 - Superexpander Cartridge..... 94
- Sound and Music..... 95

3 MACHINE LANGUAGE PROGRAMMING GUIDE..... 107

- System Overview..... 109
- Introduction to Machine Language..... 123
- Writing Your First Program..... 132

• Special Tips for Beginners.....	168
• Memory Maps.....	170
• Useful Memory Locations.....	178
• The KERNAL.....	182
• KERNAL Power Up Activities.....	211
• VIC Chips.....	212
6560 (Video Interface Chip).....	212
6522 (Versatile Interface Adapter).....	218

4 INPUT/OUTPUT GUIDE.....227

• User Port.....	229
• The Serial Bus.....	234
• Using the VIC Graphic Printer.....	236
• VIC Expansion Port.....	241
• Game Controllers.....	246
Joystick.....	246
Paddles.....	248
Light Pen.....	250
• RS-232 Interface Description.....	251

APPENDICES.....261

A. Abbreviations for BASIC Keywords.....	263
B. Screen & Border Color Combinations.....	265
C. Table of Musical Notes.....	266
D. Screen Display Codes.....	267
E. Screen Memory Maps.....	270
F. ASCII and CHR\$ Codes.....	272
G. Deriving Mathematical Functions.....	275
H. Error Messages.....	276
I. Converting Programs to VIC 20 BASIC.....	278
J. Pinouts for Input/Output Devices.....	280
K. VIC Peripherals & Accessories.....	284

INDEX.....285

SCHEMATIC.....291

INTRODUCING. . . THE PROGRAMMER'S REFERENCE GUIDE!

The Friendly Computer deserves a Friendly Reference Book. That's why we wrote the VIC 20 PROGRAMMER'S REFERENCE GUIDE . . . a book that gives you more information about your VIC 20 Personal Computer than *any other source*. This guide was compiled from the experience of Commodore's international programming staffs in more than half a dozen countries, and is designed to be used by first-time computerists as well as experienced programmers.

To cover the areas VIC 20 programmers are most interested in, we divided the book into four sections: BASIC Programming, Machine Language Programming, Input/Output Interfacing and Programming Graphics & Sound.

Here are just a few of the ways the VIC 20 Programmer's Reference Guide helps meet your programming needs:

—Our complete "dictionary" includes not only BASIC commands but also sample programs to show you how they work.

—Need an introduction to Machine Level Programming? Our laymen's overview gets you started.

—The exclusive Kernal helps assure the programs you write today won't be outdated tomorrow.

—The VIC's Interface section lets you expand your computer . . . from RS232 for telecommunications to joysticks, game paddles and lightpens.

—You'll have fun learning about the VIC's graphic, sound and music capabilities . . . including the unique "multicolor" mode.

—You'll discover POKES you never knew about, and probably PEEK into some memory locations you never knew existed.

There are lots of fascinating hours ahead of you. Let the Programmer's Reference Guide be your companion as you continue to explore your VIC 20 Personal Computer System.

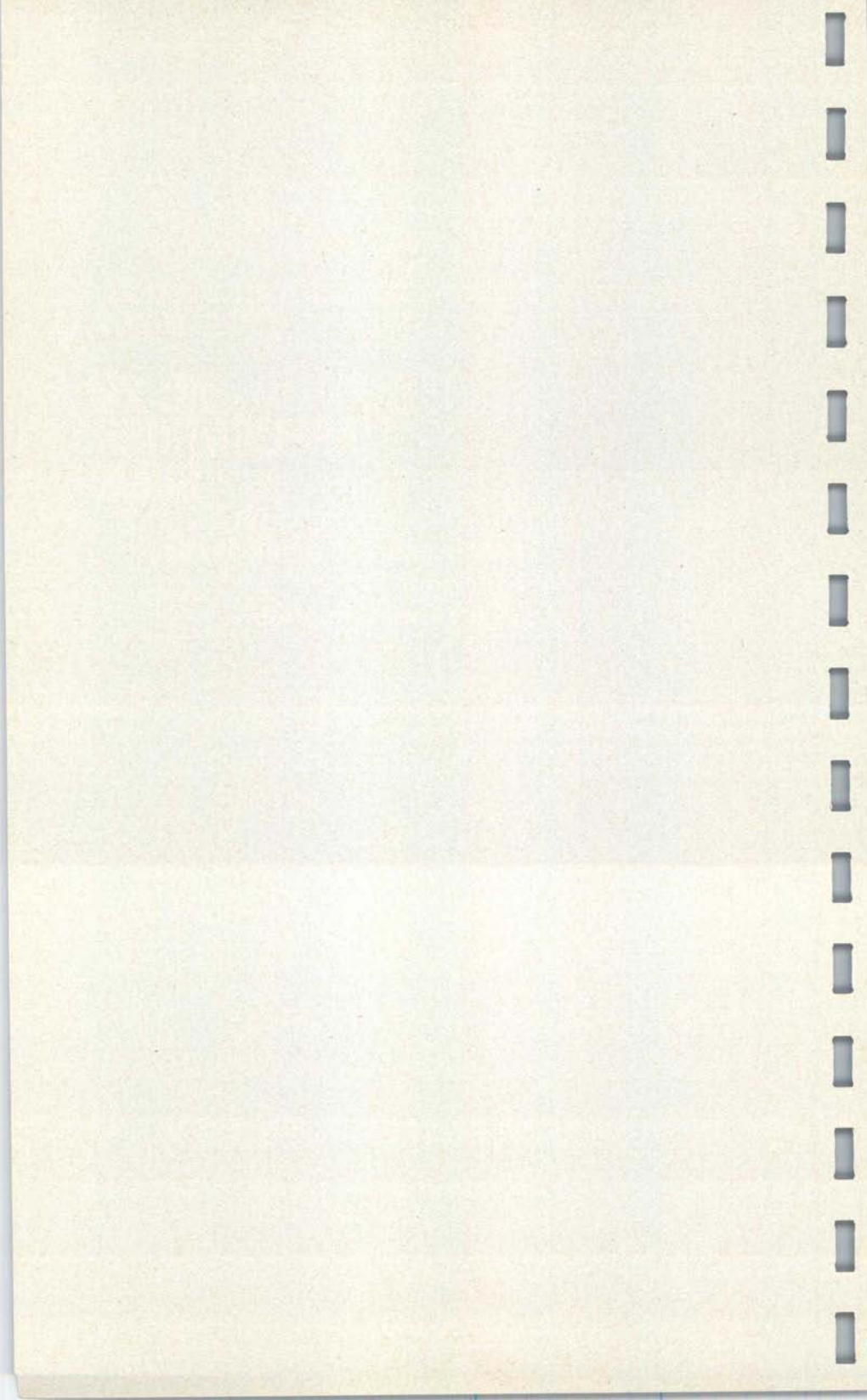
And . . . if you find any errors in this book, please send us a postcard or letter in care of VIC PROGRAMMER'S REFERENCE GUIDE, VIC Product Marketing Group, Commodore Business Machines, Inc., 681 Moore Road, King of Prussia, PA 19406. We'd appreciate your assistance in helping us "debug" our reference guide for future printings.

Enjoy your new reference guide . . . and happy programming!

—The Authors

VIC 20 APPLICATIONS GUIDE

- More than 30 applications
for your VIC 20
Personal Computer



VIC 20 APPLICATIONS GUIDE

When you first considered buying a computer, the chances are you said something like, "I know computers are good things to have and it's nice that they're finally affordable, but . . . what can I do with one?"

The great thing about a computer is that you can tailor the machine to do *what you want it to*—you can make it calculate your home budget, play arcade-style action games—you can even make it talk! And the best thing is, if your VIC 20 does only ONE of the things listed below, it's well worth the price you paid for it.

Here then, is a list of applications for your VIC 20—in case you've asked yourself, "Yes, but what *else* can I do with it?"

APPLICATION	COMMENTS/REQUIREMENTS
ADVENTURE GAMES	COMMODORE provides 5 Scott Adams Adventure games on cartridge, decoded to "talk" with the VOTRAX "Type N Talk"™.
ADVERTISING & MERCHANDISING	Hook the VIC to a television and put it in a store window with an animated message flashing and you've got a great point of purchase store display.
ANIMATION	The VIC is well-suited to screen animation . . . a special aid called THE PROGRAMMABLE CHARACTER SET & GAMEGRAPHICS EDITOR is available from COMMODORE on tape cassette.
BABYSITTING	The VIC HOME BABYSITTER cartridge can keep your child occupied for hours and teach keyboard symbols, special learning concepts and relationships. A "first" from COMMODORE.

**BASIC
PROGRAMMING**

The VIC owner's guide and the TEACH YOURSELF PROGRAMMING series of books and tapes are excellent starting points. A PROGRAMMERS AID CARTRIDGE is available from COMMODORE.

**BIORHYTHM
CHARTING**

COMMODORE'S Biorhythm program on tape has a special compatibility feature which lets you compare yourself to anyone else by simply typing in your birthdates.

CHESS GAME

SARGON II (on cartridge from COMMODORE) has been called the most powerful microcomputer chess program anywhere.

COLLECTIONS

COMMODORE will provide a cartridge which allows collectors to record their collections (stamps, coins or other items) on tape or floppy diskettes, and print out these lists on the VIC GRAPHIC PRINTER.

COMMUNICATION

VICMODEM™, VICNET™ and VIC-TERM™ are all products which allow VIC owners to communicate by telephone with other computer owners, or telecomputing services like CompuServe™ or The Source™.

**COMPOSING
SONGS**

The VIC's 3 tone generators cover 5 octaves and may be used to write and record music. The best music writing accessory is the SUPEREXPANDER CARTRIDGE which lets you write music in note form and save it on tape or disk.

DEXTERITY

Hand-to-eye coordination and manual dexterity are aided by several of COMMODORE's VIC games . . . including the "Jupiter Lander" and night driving simulations, among others.

EDUCATION

The *COMMODORE Educational Computing Resource Book* contains information on educational uses of computers in general as well as educational software lists for the VIC 20. Available through COMMODORE computer dealers.

EXPENSE RECORDS

A CALENDAR/EXPENSE RECORD tape is offered by COMMODORE.

FOREIGN LANGUAGE

The VIC Programmable Character Set Editor lets any user replace the VIC character set with user-defined foreign language characters.

FORMULA/FIGURES

The VIC has the same powerful math routines built into its operating system as the COMMODORE PET/CBM microcomputers. Complex formulas may be calculated quickly and easily either directly or under program control (see the OPERATORS section of the VIC user manual and/or the matching section in this book).

GAMBLING

COMMODORE provides several games which provide hours of gambling fun without risking any money . . . programs like VIC21 Casino Style Blackjack (on tape), SUPERSLOT (cartridge) and DRAW POKER (cartridge).

GAMES

Everything from space games on cartridge to Blackjack on tape, plus REAL ARCADE games adapted from the most popular coin-operated games in the world.

GRAPHICS PLOTTING

The SUPEREXPANDER CARTRIDGE offers 3K memory expansion, hi-resolution multi-color graphics plotting, easy function key definition, and musicwriting commands . . . all in one cartridge.

HOME INVENTORY

The HOME INVENTORY tape in COMMODORE's HOME CALCULATION SIXPACK provides a low priced method for storing and updating lists of belongings for insurance purposes, business purposes, etc.

INSTRUMENT CONTROL

The VIC has a serial port, RS-232 port and IEEE-4888 adapter cartridge for use in a variety of special industrial applications.

JOURNALS OR CREATIVE WRITING

The VIC is excellent for making daily journal entries, using the VIC TYPEWRITER or VICWRITER. Information can be stored on the VIC DATASSETTE tape recorder or VIC DISK DRIVE, and printed out on a VIC GRAPHIC PRINTER.

LIGHTPEN CONTROL

Applications using a lightpen to specify items can use any commercial lightpen which fits the VIC game port connector . . . at least two makers market lightpens which work with the VIC.

LOAN/MORTGAGE CALCULATION

Try the LOAN/MORTGAGE CALCULATOR from COMMODORE.

MACHINE CODE PROGRAMMING

COMMODORE's *PROGRAMMER'S REFERENCE GUIDE* includes a machine language section. The VICMON™ machine language monitor cartridge is recommended. VIC machine language programs may also be written in assembly language on the PET/CBM using the COMMODORE Assembler Development System.

MATH PRACTICE TOOL

Several software companies offer educational programs on tape for the VIC. COMMODORE's first math practice program, called "SPACE-MATH," is available on tape.

NETWORKING & DISTRIBUTED PROCESSING

Networking may be achieved by using the VIC as part of a telephone or RS-232 hookup, or by using a commercially available network system.

PAYROLL & FORMS PRINTOUT

The VIC can be programmed to handle a variety of entry-type business applications. Upper/lower case letters combined with VIC "business form" graphics make it easy to design forms, which can be easily printed out on the VIC GRAPHIC PRINTER.

PERSONAL BUDGET

COMMODORE provides PERSONAL FINANCE programs on tape and on plug-in cartridge.

PORTFOLIO ANALYSIS

Business software which performs this function is available as printed programs in books available from most computer stores. This service is also available through telecomputing services.

PRINT INFORMATION ON PAPER

The VIC GRAPHIC PRINTER prints letters, numbers and graphics in high quality dot matrix format. RS-232 PRINTERS including letter quality printers may also be used with the proper interfacing. An IEEE 488 INTERFACE cartridge may also allow IEEE printer use.

RECIPES

See the recipe program called "MIKE'S CHICKEN SOUP" in the VIC owner's manual, or check a computer book rack (most "practical" program books contain recipe programs).

SIMULATIONS

Computer simulations permit dangerous or expensive experiments to be performed at minimum risk & expense.

SPORTS DATA

The Source™ and CompuServe™ both provide sports information.

STOCK QUOTES

The VIC, a modem, and a subscription to The Source™ or Compu-Serve™ can cost less than \$500.

TALKING VIC

Connect the VIC to a voice synthesizer such as the "Type N Talk"™ manufactured by VOTRAX INC.

TERM PAPERS & REPORTS

The VIC helps students research current library-type sources over the telephone . . . and compose, edit and print out their reports on the VIC and VIC GRAPHIC PRINTER . . . the same type of computer services which were previously available only through large institutions at a cost of many thousands of dollars.

TERMINAL & MODEM

VIC accessories include an RS-232 modem interface (for use with RS-232 modems) or the ultra low-priced VICMODEM™

TYPING PRACTICE

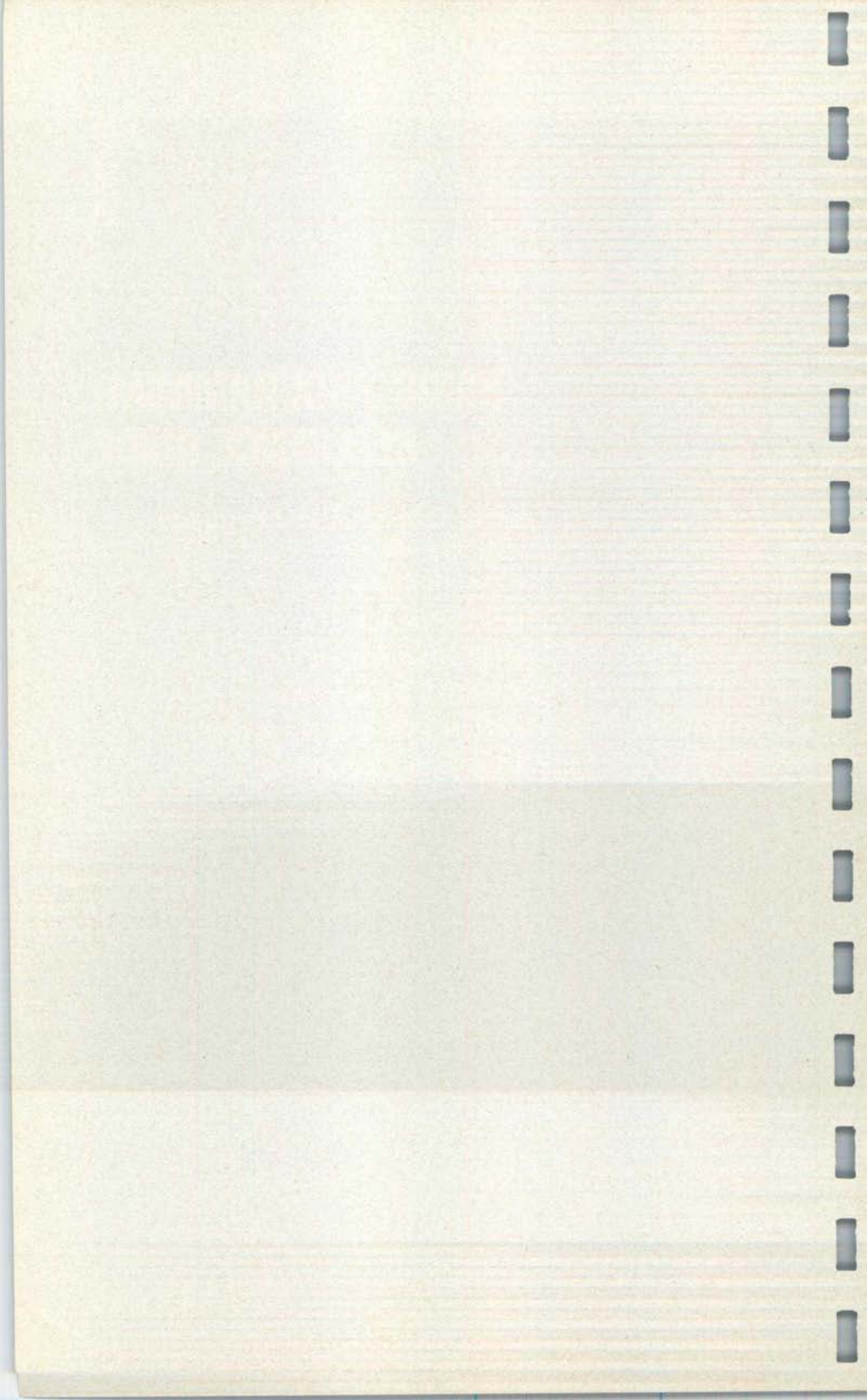
The reverse side of the VIC TYPEWRITER has a "TYPING TUTOR" program.

WORDPROCESSING

THE VIC TYPEWRITER™ is available on tape and the VICWRITER™ cartridge also provides wordprocessing power. Both work with the VIC GRAPHIC PRINTER.

BASIC PROGRAMMING REFERENCE GUIDE

- **VIC BASIC: The Language of the VIC**
- **Commands**
- **Statements**
- **I/O Statements**
- **BASIC Functions**
- **Numbers and Variables**
- **Logical Operators**



VIC BASIC: THE LANGUAGE OF THE VIC

The BASIC computing language is a powerful and easy-to-use means of communicating instructions to your VIC 20 Personal Computer. VIC BASIC is the same language used in the Commodore PET/CBM line of microcomputers, and is nearly identical to the BASIC used in most other personal computers. Learning BASIC now can prepare you to move up to a more sophisticated computer in the future, and can also give you the foundation you need to learn other "higher level" computing languages.

If you're a first-time computerist, you'll be pleased to know you can write your first BASIC program on the VIC within 15 minutes, using the VIC 20 PERSONAL COMPUTER GUIDE which comes with the machine. Additional self-teaching aids are available from Commodore as part of the TEACH YOURSELF PROGRAMMING SERIES, and classes offered by schools, computer centers and retail stores can give you a solid grounding in the fundamentals of BASIC within 4-6 hours.

The VIC BASIC instructions which follow will provide a valuable reference as you learn to write BASIC, or as you put into practice the techniques you've already learned. Each entry in the listing explains how the instruction is used, with practical examples. Additional programming tips are included in a separate "BASIC PROGRAMMING TIPS" section.

BASIC has approximately 60 words in its vocabulary and is surprisingly easy to learn. That doesn't mean you can't keep improving, however. Like any language, BASIC has its own "idioms" and complexities which you can use to write increasingly sophisticated programs. VIC BASIC even has a sort of "slang" in that you can *abbreviate* most of the commands by typing the first letter of the instruction and the SHIFTED second letter. Using abbreviated commands to write programs makes programming the VIC fast and convenient. (Note that if you LIST a program written in abbreviated form, the full-length commands are displayed to help you read your program.)

In BASIC, all instructions are commonly referred to as "commands," although technically the BASIC instruction set can be broken down into several areas . . . which is how we've grouped them in the following VIC BASIC "vocabulary" guide. We've included separate sections on several types of BASIC instructions: Commands, Statements, Input/Output Statements, Functions, Numbers and Variables, and Operators.

It should also be noted here that while we communicate with the VIC through the BASIC language, the VIC's "native" vocabulary is *Machine Language* which is based on a binary or hexadecimal numbering system. BASIC is really a *translation* of Machine Language into terms we humans can understand . . . which is why BASIC programs generally run slower than Machine Language programs, since BASIC programs have to be interpreted into Machine Language before they run, while Machine Language programs run immediately. See "Introduction to Machine Language Programming" for more information.

Of course, you don't have to know BASIC to take advantage of the VIC's computing power . . . you don't even have to know how to type. We like to say that with the VIC 20, the first thing you do is learn "computing" . . . not "programming." You don't have to be an auto mechanic to drive a car, and by the same token you don't have to be a programmer to "drive" your VIC 20. Still, knowing something about how your car works mechanically helps you maintain and use your car to best advantage. Likewise, knowing how the computer is programmed helps you get the most out of your VIC 20.

In the future, being able to "speak" a computer language will give you a tremendous advantage over those who can't . . . not because you can write a computer program, but because you'll have a better understanding of what a computer is and does, and you'll be able to make better use of computing at school, on the job and at home. Learning BASIC . . . or at least how it works . . . will bring you close to the future and prepare you for the dramatic technological changes that are already occurring as part of the "Computer Revolution."

TO AVOID CONFUSION, PLEASE NOTE:

O = letter O as in OPEN

0 = ZERO

I = letter I as in INPUT

1 = number ONE

* = type the large asterisk key

Format = all format entries shall be typed in on one line

[] = information contained inside brackets in the Format lines is *optional*.

WARM START—If you get into trouble or want to break out of a program while it's running, you can hold down the RUN/STOP key and hit the RESTORE key. This combination resets the VIC *without losing your program*. Now you can LIST or RUN your program again.

COMMANDS

BASIC commands tell the VIC to do something with a *program*. For example, you "command" the VIC to list, run, stop, continue, save and load programs.

BASIC commands may be used in direct mode (without line numbers) or as instructions in BASIC programs. In direct mode, commands are executed as soon as the RETURN key is pressed.

In a BASIC program, commands are included like any other instruction, and executed when you type RUN. The only command which may not be used in a program is CONT.

VIC BASIC commands include the following:

CONT
LIST
LOAD
NEW
RUN
SAVE
VERIFY

CONT

Format:

CONT

Abbreviation:

C  O

Screen Display:


C ☐

This command is used to re-start the execution of a program which has been stopped by either using the STOP key, a STOP statement, or an END statement within the program. The program will re-start at the exact place from which it left off. While the program is stopped, the user can inspect or change any variables or look at the program. CONT will not work if you have changed or added lines of the program (or even just moved the cursor to a program line and hit RETURN without changing anything), or if the program halted due to an error, or if you caused an error before typing to re-start the program. The message in this case is CAN'T CONTINUE ERROR.

This is a handy tool when debugging a program. It lets you place STOP statements at strategic locations in the program, and examine variable values when the program stops. You can keep using STOP and CONT until you find what you're looking for.

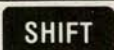

EXAMPLE:

```
10 PI = 0: C = 1
20 PI = PI + 4/C - 4/(C + 2)
30 PRINT PI
40 C = C + 4:GOTO 20
```

This program calculates the value of PI. RUN this program, and after a short while hit the  key. You will see the display:
BREAK IN 20



Type the command PRINT C to see how far the VIC has gotten. Then use CONT to resume from where the VIC left off.

LIST

<u>Format:</u> *	<u>Abbreviation:</u>	<u>Screen Display:</u>
LIST [from line number] - [to line number]	L  I	L 

The LIST command allows you to look at lines of the BASIC program currently in the VIC's memory. The VIC's powerful screen editor allows you to easily and quickly edit programs which you have LISTed.

The LIST command can specify which program line numbers will be shown. If you type LIST and a single line number, only that line will be displayed. If the number is followed by a hyphen (-), all the lines from that number forward will be shown. If the number is *preceded* by the hyphen, all lines from the beginning to that line will be shown. You can LIST a *range* of line numbers by typing the two line numbers separated by a hyphen, in which case the lines from the first number to the second number are shown. If LIST 0 is typed, the whole program will be LISTed and not just line 0.

If the program length exceeds the length of the screen display, the first lines in the program will scroll off the screen during LISTing. To slow down the scrolling, hold down the  key. To stop the program during a LIST, hit the  key.

The LIST command may also be used as a statement within a BASIC program, but the program will stop as soon as the LIST is finished.

*All format entries should be typed in one line.

EXAMPLES OF LIST COMMAND

LIST	LISTs whole program
LIST 0	LISTs whole program
LIST 100	Line 100 only
LIST 100-	Everything from 100 on
LIST -100	Shows from start to 100
LIST 100-150	Starts at 100 and stops with 150

EXAMPLE OF LIST STATEMENT

(Program Mode)

```
10 PRINT"THIS IS LINE 10"  
20 LIST  
30 PRINT"THIS IS LINE 30"
```

LOAD

Format:

LOAD ["filename",
device, command]

Abbreviation:

L **SHIFT** O

Screen Display:

L ☐

The LOAD command transfers a program from cassette tape or disk into the VIC's memory, where it can be used or changed.

LOAD FROM TAPE

If the program to be LOADED is the first one on the tape, all you have to type is the word LOAD by itself. Unless the PLAY key was already down, the VIC answers with the message:

PRESS PLAY ON TAPE.

Once the recorder has been started, the VIC says:

OK
SEARCHING
FOUND
LOADING

At this point, any program that had been in memory is lost, because the new one has begun to take its place. (If you use the **RUN STOP** key to halt the LOAD, there will likely be a spot in the program with garbage, just at the point where you stopped it.)

Once the program has finished the LOAD, the VIC says:

READY.

When the program is not the first on the tape, or you're not sure that it is, the VIC can search for the program you want. Type LOAD and the name of the program inside quote marks ("), or the name of a string variable containing the name of the program. The VIC will show any other programs or files that it sees on the tape with the message:

FOUND name

The VIC will only LOAD the correct program, and will not LOAD a data file on that tape.

LOAD FROM DISK

In order to bring in a program from a device other than the tape, a device number is used. Following the name of the program, type a comma (,) and the number (or variable containing the number). The cassette is device number 1. The disk drive is device number 8. See the manual of the device for its number.

If the program with that name is not found on the device, a FILE NOT FOUND ERROR will result. This doesn't happen on tape, since the VIC has no way of sensing that there are no more programs on a tape. It is possible to put an end-of-tape marker on the tape, using the SAVE or OPEN statements, and if the VIC reads this marker while searching a tape, an error message appears.

The VIC will automatically LOAD the program into the beginning of BASIC program memory, at location 4096 in a machine without an extra 3K of memory, or at 1024 with the extra 3K. For certain applications this may not be convenient. By following the device number with a comma and the command number 1, the VIC will be sure to LOAD the program in the same spot in memory from which it was SAVED.

LOAD can be used as a statement in a BASIC program. The program will be RUN as soon as LOADING is finished. Variables used in the first program will not be cleared as long as the new program is shorter in length than the older one. If a longer program is LOADED on a short one, the BASIC program lines will over-write the variables, and changing a variable will mess up the program.

Note that using an asterisk can save loading time (see examples).

EXAMPLES:

LOAD	Reads in the next program from tape.
LOAD "HELLO"	Searches tape until the program called HELLO is found, then it is LOADED.
LOAD A\$	Uses the name in A\$ to search.

LOAD "*", 8	LOADs first program from disk.
LOAD "AB*", 8	Loads first program beginning with AB.
LOAD "HELLO", 8	Looks for a program on device 8 (disk drive).
LOAD "",1,1	Looks for the first program on tape, and LOADs it into the same part of memory that it came from.
10 LOAD"NEXT", 8	Finds the program called NEXT on device 8, LOADs, then RUNs it.

Because of possible problems with old tapes or misaligned recorders, it is possible that the program will not LOAD correctly. The VIC stores two copies of the program on the tape. If they don't match, the message LOAD ERROR is displayed. The program may LIST correctly, but probably won't. In any case, there is most likely some problem, some section of memory that is not right, and the program should be re-LOADed. It is wise to make an extra copy of any program, in case you run into this problem at some time.

In the case of a very bad LOAD, some of the important memory locations inside the VIC may be changed. If you get weird results after a LOAD, like the VIC not understanding normal BASIC commands anymore, you'll have to turn the VIC off and then on again.

NEW

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
NEW	None	None

NEW is used to tell the VIC to erase a current program from memory so a different program can be used. Unless the program is stored (on tape or disk), it will be lost unless it is typed in again from the beginning. For this reason, you should BE CAREFUL when using this command! Not clearing out an old program before typing in a new one can result in a confusing mixing of the two programs.

NEW can also be used as a statement within a program. When this instruction is executed, the program in memory is erased, and the program stops. This is not a good programming technique, especially if you RUN the program and it erases itself while you're writing or debugging it.

EXAMPLE:

- NEW Clears the program & variables.
 10 NEW Performs the NEW operation and stops the program.

RUN

Format:

RUN [line number]

Abbreviation:

R **SHIFT** U

Screen Display:

R 

This command causes a BASIC program to begin operating. The command RUN by itself starts the program at the lowest numbered line. All variable values are cleared when this command is given.

RUN followed by a number causes the program to start working from another line than the lowest numbered one. If that line number does not exist, the message UNDEF'D STATEMENT ERROR appears. RUN followed by a variable will first clear the value of that variable, and try to start the program at line 0, if it exists.

RUN can also be used as a statement within a program. Keep in mind that all variables are cleared when this statement is executed.

EXAMPLES:

RUN Starts at the beginning
RUN 100 Starts at line 100
RUN X Starts at line 0, or UNDEF'D
STATEMENT ERROR if no line 0

SAVE

Format:

SAVE ["filename",
device, command]

Abbreviation:

S **SHIFT** A

Screen Display:

S 

The SAVE command stores a program currently in memory on tape or disk. The program being SAVED is not affected and remains in the VIC's memory after the save operation. Programs on tape are stored twice automatically, so the VIC can check for errors when LOADING the program back in.

The command SAVE all by itself sends the program to the cassette deck without a name. When the command is given, the VIC will say:

PRESS RECORD AND PLAY ON TAPE

Holding down the RECORD button, press PLAY, and the VIC will say:

OK
SAVING

and begin storing. The VIC cannot check the RECORD key; it can only sense that the tape is moving, so be sure to press RECORD. If PLAY was already pressed, no message appears.

When the program has been SAVED, the VIC will give the message:

READY.

The VIC has no way of searching for a blank spot on the tape, but just records wherever it is, erasing any information that may have been there. However, the VERIFY command can be used to find the end of the last program.

SAVE can be followed by a program name in quotes or in a string variable. The VIC will then write the program name before the program on the tape, which lets the VIC find it more easily.

The program can be SAVED on a device other than the tape deck by giving a device number. To do this, put a comma after the program name, and then the number of the device. The cassette deck is device number 1, and the disk is number 8. (See examples.)

It is possible to instruct the VIC to SAVE a program so it will not be moved in memory when LOADED. Using command number 1 after the device number will do this. This is useful when working with different memory configurations which may cause VIC memory locations to shift.

To prevent a user from accidentally trying to read past the last information on the tape, the VIC can also write an end-of-tape marker after the program. To do this, follow the device number with a comma and command number 2. When the VIC finds this marker, it will stop and show the message DEVICE NOT PRESENT ERROR.

Command number 3 is a combination of 1 and 2, telling the program on tape not to relocate and to put an end-of-tape marker after the program.



SAVE can also be used as a statement within a BASIC program. When this statement is hit, the program will be SAVED normally, with the usual prompts appearing on the screen. The program resumes normally after the SAVE.

EXAMPLES

SAVE	Stores program on tape without name
SAVE "HELLO"	Stores on tape with name HELLO
SAVE A\$	Stores on tape with name in A\$
SAVE "HELLO",8	Stores on device number 8 (disk drive)
SAVE "HELLO",1,1	Won't relocate HELLO upon re-LOADing

SAVE "HELLO",1,2	Puts an end-of-tape marker after the program.
SAVE "HELLO",1,3	Won't relocate & end-of-tape marker
10 SAVE"HELLO"	Saves the program, then goes on with the next program line.

VERIFY

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
VERIFY " <u>filename</u> ", <u>device</u>	V  E	V 

This command checks the program on tape or disk against the program in the VIC's memory. VERIFY is normally used right after a SAVE, to make sure the program was stored correctly on the tape or disk. VERIFYing a program after it has been LOADED is useless, since the same incorrect program could be in both places.

When you tell the VIC to VERIFY, this message shows:

PRESS PLAY ON TAPE

Once the tape is moving, the VIC checks the program against memory, reading until the end of the program. If the copies match, the VIC says:

OK
READY.

If there is a problem, you will see this message:

?VERIFY ERROR

In this case, you should immediately SAVE the program on a different tape or disk and try again.

The format of the VERIFY command is similar to the LOAD command. A program name can be given either in quotes or a string variable, and the VIC will search for that program. If a comma and a device number follow the name, the VIC will look at the device designated.

VERIFY is also used to position a tape just past the last program, so a new program can be added to the tape without over-writing an older one. Just VERIFY with the name of the last program there; the VIC searches, checks the program, and stops with a VERIFY ERROR. However, your program is still in memory, and now the tape is at a blank spot. You can SAVE without worry.

EXAMPLES:

VERIFY	Checks the first program on tape.
VERIFY "HELLO"	Searches for HELLO, then checks
VERIFY "HELLO",8	Looks on device 8 for the program.

STATEMENTS


CLR
DATA
DEF FN
DIM
END
FOR ... TO ... STEP
GET
GOSUB
GOTO or GO TO
IF ... THEN
INPUT
LET
NEXT
ON
POKE
PRINT
READ
REM
RESTORE
RETURN
STOP
SYS
WAIT

CLR

Format:


CLR

Abbreviation:

C  L

Screen Display:

C 


This statement clears out any variables that have been defined, un-DIMensions any arrays, and RESTOREs the DATA pointer back to the beginning. This makes available all the RAM memory that the variables had used so that it can be used for something different. The RUN command automatically performs the CLR operation, as does LOADING a new program or doing a NEW. Don't confuse this statement with the  key, which clears the screen.

EXAMPLE:

```
100 A = 1014
110 CLR
120 PRINT A
```

When RUN, this program PRINTs a zero on the screen.

DATA

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
DATA value [, value, . . . , value]	D SHIFT A	D 


The DATA statement holds information that will fill variables in a READ statement. Any type of information can be stored here, separated by commas. If a comma, space, or colon are to be used as data, they must be enclosed in quote marks (""). Two commas with nothing in between will be read as zero, or an empty string. Note: the information in brackets is *optional*.

The line containing the DATA statement doesn't actually have to be executed while RUNning the program, so most programmers leave all the DATA statements at the end of the program, out of the way.

EXAMPLE:

```
10 READ A : PRINT A
20 READ A, B, C : PRINT A; B; C
30 READ A$ : PRINT A$
40 FOR L = 1 to 5 : READ A : PRINT A; : NEXT
50 READ A$, A, B, B$ : PRINT A$, A, B, B$
60 END : REM YOU NEVER ACTUALLY HAVE TO HIT THE
DATA STATEMENTS!
960 DATA 1
970 DATA 1,2,3
980 DATA ABC
990 DATA 2,3,5,7,11,"HELLO" -1.2445E-5, 69.7767, "A, B, C"
```

DEF FN

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
DEF FN [name] (variable) = formula	D SHIFT E	D 

When a long mathematical formula is used several times in different lines of a program, program memory and typing time can be saved by using a defined function for the formula. The function is then used throughout the program in place of the lengthy formula.

The name of the function will be the letters FN and any variable name you choose, one or two letters long. The DEF FN statement must be executed at least once for the program to use it, so this statement is normally placed at the beginning of the program.

The function name is followed by a variable name inside parentheses. Next comes an equal sign, and then the formula.

Here's an example of a simple formula definition, with an example of its use in the program.

EXAMPLE 1:

```
10 DEF FNA(X) = 7 * X
20 PRINT FNA(1)
30 PRINT FNA(3)
```

The result of line 20 is 7, and the result in line 30 is 21.

EXAMPLE 2:

```
10 DEF FNA(X) = INT(RND(1)*6) + 1
20 PRINT FNA(10)
```

When the function in this example is used in the program, the value of the number in parentheses in line 20 doesn't have any effect on the result. This is because in line 10 the variable (X) in the parentheses doesn't appear in the formula on the right.

The next example does use the variable name in the formula.

EXAMPLE 3:

```
10 DEF FNA(X) = INT(RND(1)*X) + 1
20 PRINT FNA(10)
```

In this case, the number in the parentheses in line 20 does affect the result. The number in the parentheses in line 20 is the largest random number that will be picked.

The result of a defined formula must always be a number; there are no defined functions for string variables.

DIM

Format:

DIM variable

(number, . . . , number),

[variable (number, . . . , number), . . .]

Abbreviation:

D **SHIFT** I

Screen Display:

D 

This statement defines an array or matrix of variables, which allows you to use the variable name with a subscript. The subscript points to the element in the array being used. The lowest element number in an array is zero, and the highest is the number given in the DIM statement. If an array variable is used without a DIM statement to create it, it is automatically DIMensioned to 10 in each dimension.

Let's suppose we wanted to keep track of the score of a football game. There are 2 teams, and four quarters plus a possible overtime quarter in the game. We could use a matrix to hold the scores in each quarter. Here is a program that asks you for the score of each team in each quarter:

EXAMPLE:

```
100 DIM S(1,5) , T$(1)
110 INPUT "TEAM NAMES" ; T$(0), T$(1)
120 FOR Q = 1 TO 5
130 FOR T = 0 TO 1
140 PRINT T$(T), "SCORE IN QUARTER" Q
150 INPUT S(T,Q)
160 S(T,0) = S(T, 0) + S(T, Q)
170 NEXT T,Q
180 PRINT CHR$(147) "SCOREBOARD"
190 PRINT "QUARTER";
200 FOR Q = 1 TO 5
210 PRINT TAB(Q*2 +9) Q;
220 NEXT
230 PRINT TAB (15) "TOTAL"
240 FOR T = 0 TO 1
250 PRINT T$(T);
260 FOR Q = 1 TO 5
270 PRINT TAB (Q*2 +9) S(T, Q);
280 NEXT
290 PRINT TAB(15) S(T,0)
300 NEXT
```

The element numbers in every dimension start at 0 and end at the number in the DIM statement. The number of elements created in any dimension is the maximum subscript number PLUS 1. The total number of elements is equal to the product of the number of elements in all dimension multiplied together.

There may be any number of dimensions and any number of elements in an array, limited only by the amount of RAM memory that is available to hold the variables. The array may be made up of normal numeric variables, as shown above, or of strings or integer numbers. If the variables are to be other than normal numeric, simply use the \$ or % signs after the variable name to indicate string or integer variables.

It's easy to calculate the amount of memory that will be used up by an array:

MEMORY USED = 5 bytes for variable name
2 bytes for each dimension
2 bytes/element for integer variables
5 bytes/element for normal numeric variables
3 bytes/element for string variables
1 byte for each character in each string element

END

Format:

END

Abbreviation:

E  N

Screen Display:

E 

This statement will finish the program when RUNning and return complete control of the VIC to the person operating it. The CONT command can be used to resume execution of the program after the END statement was reached, because no variables or pointers are cleared.

The END statement results in the message: READY.

The difference between STOP and END statements is slight: the STOP statement displays the message:

BREAK IN LINE XXX

Neither STOP nor END is required to appear at any point in the program in VIC BASIC, because a program running out of lines to execute will END all by itself.

FOR ... TO ... STEP ...

Format:

FOR variable =

Abbreviation:

F **SHIFT** O

Screen Display:

F ☐

start TO limit [STEP increment]

This is a special BASIC statement that lets you easily use a variable as a counter. You must specify certain parameters: the variable name, its starting value, the limit of the count, and how much to add during each cycle.

Here is a simple BASIC program that counts from 1 to 10, PRINTING each number and ENDing when complete, and using no FOR statements:

```
100 L = 1
110 PRINT L
120 L = L + 1
130 IF L <= 10 THEN 110
140 END
```

Using the FOR statement, here is the same program:

```
100 FOR L = 1 TO 10
110 PRINT L
120 NEXT L
130 END
```

As you can see, the program is shorter and easier to understand using the FOR statement. Here is a closer look at the parameters, to see how everything works.

The variable can be any numeric variable name except an array variable. When the program reaches a FOR statement, variable is set to the value of start. The program proceeds with the statements, until a statement containing the word NEXT is reached.

At that point, increment is added to variable's value. The STEP is optional, and if there is no STEP shown increment is assumed to be +1.

After increment has been added to variable, the value of variable is compared to limit. If the limit has not been exceeded the program continues with the line after the FOR statement. If the limit has been passed, the line to be executed is the line following the NEXT statement. Note: if the STEP value is positive, variable will exceed limit when its value is greater than limit, and if the STEP value is negative, the variable must be less than limit to end the count. The loop will always be executed at least once, regardless of the values in "start" and "limit".

EXAMPLE:


```
100 FOR L = 100 TO 0 STEP -1
100 FOR L = PI TO 6*PI STEP .01
100 FOR AA = 3 TO 3
```

GET

Format:

GET variable

Abbreviation:

G  E

Screen Display:

G 

This statement lets you input one character at a time from the keyboard. Whatever character was hit goes into the variable. If no key was pressed, a zero is placed in a numeric variable, or an empty value ("") in a string variable. This differs from the INPUT statement in one major respect: if no key is typed, the program continues running here, and in the INPUT statement it waits for the user to type something.

The GET statement is usually placed in a loop to wait for the keystroke.

EXAMPLE 1:

```
10 GET A$: IF A$ = "" THEN 10
```

The GET can also be used to allow the program to continue processing while waiting for data. Example 2 is a simple GET editor with a blinking cursor.

EXAMPLE 2:

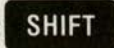
```
10 C = 0 : Q = 18
20 GET A$ : C = C + 1
30 IF C = 10 THEN Q = 164 - Q : C = 0
40 PRINT CHR$(Q) CHR$(32) CHR$(146) CHR$(157);
50 PRINT A$; : GOTO 20
```

GOSUB

Format:

GOSUB line number

Abbreviation:

GO  S

Screen Display:

GO 

This is a specialized form of the GOTO statement, with the important difference that GOSUB remembers where it came from. When the RETURN statement (different from the RETURN key on the keyboard) is reached in the program, the program jumps back to the statement immediately following the original GOSUB statement from which it came.

The major use of a subroutine (GOSUB really means GO to a SUBroutine) is when there is a small section of program that is used by different sections of the program. By using subroutines rather than repeating the same lines over and over at different places in the program, you can save lots of program space. In this way, GOSUB is similar in use to DEF FN: DEF FN lets you save space when using a formula, and GOSUB saves space when using a several-line routine.

Here is an inefficient program that doesn't use GOSUB:

```
100 PRINT "THIS PROGRAM PRINTS"  
110 FOR L = 1 TO 500 : NEXT  
120 PRINT "SLOWLY ON THE SCREEN"  
130 FOR L = 1 TO 500 : NEXT  
140 PRINT "USING A SIMPLE LOOP"  
150 FOR L = 1 TO 500 : NEXT  
160 PRINT "AS A TIME DELAY."  
170 FOR L = 1 TO 500 : NEXT
```

Here is the same program using GOSUB:

```
100 PRINT "THIS PROGRAM PRINTS"  
110 GOSUB 200  
120 PRINT "SLOWLY ON THE SCREEN"  
130 GOSUB 200  
140 PRINT "USING A SIMPLE LOOP"  
150 GOSUB 200  
160 PRINT "AS A TIME DELAY."  
170 GOSUB 200  
180 END  
200 FOR L = 1 TO 500 : NEXT  
210 RETURN
```

Each time the program executes a GOSUB, the line number and position in the program line are saved in a special area called the "stack," which takes up 256 bytes of your memory. This limits the amount of data that can be stored in the stack. Therefore, the number of subroutine return addresses that can be stored is limited, and care should be taken to make sure every GOSUB hits the corresponding RETURN, or else you'll run out of memory even though you have plenty of bytes free.

GOTO or GO TO

Format:

GOTO line number

Abbreviation:

G **SHIFT** O

Screen Display:

G ☐

This simple statement allows the BASIC program to execute lines out of numerical order. The word GOTO followed by a number will make the program jump to the line with that number. GOTO cannot be followed by a variable, but must have the line number typed after the word GOTO.

EXAMPLE 1:

```
10 GOTO 10
```

Notice that the loop in the example never ends, since the program keeps running the same line over and over. This is called an "infinite loop," and can be utilized when you want a program to stop in place and wait. The only way to stop an infinite loop is with the **RUN STOP** key.

EXAMPLE 2:

```
10 PRINT "HELLO";  
20 GOTO 10
```

IF . . . THEN

Format Choices:

IF expression THEN line number

IF expression THEN statement

Abbreviation: Screen Display:

None

None

This is the statement that gives BASIC most of its "intelligence," the ability to evaluate conditions and take different actions depending on the outcome.

The word IF is followed by an expression, which can include variables, strings, numbers, comparisons, and logical operators. The word THEN is followed *on the same line* by either a line number or one or more BASIC statements. When the expression is false, everything after the word THEN on that line is ignored, and

execution continues with the next line number in the program. A true result makes the program either branch to the line number after the word THEN or execute whatever other BASIC statements are found on that line.

EXAMPLE 1:

```
100 INPUT "TYPE A NUMBER"; N
110 IF N <= 0 THEN 200
120 PRINT "SQUARE ROOT=" SQR(N)
130 GOTO 100
200 PRINT "NUMBER MUST BE >0"
210 GOTO 100
```

This program prints out the square root of any positive number. The IF statement here is used to validate the result of the INPUT. When the result of $N \leq 0$ is true, the program skips to line 200, and when the result is false the next line to be executed is 120. Note that GOTO is not needed with IF . . . THEN, as in line 110 where THEN 200 actually means THEN GOTO 200.

EXAMPLE 2:

```
100 FOR L = 1 TO 100
110 IF RND(1) < .5 THEN X = X + 1 : GOTO 130
120 Y = Y + 1
130 NEXT L
140 PRINT "HEADS= " X
150 PRINT "TAILS= " Y
```

The IF in line 120 tests a random number to see if it is less than .5. When the result is true, the whole series of statements following the word THEN is executed: first X is incremented by 1, then the program skips to line 130. When the result is false, the program drops to the next statement, line 120.

EXAMPLE 3:

```
100 PRINT CHR$(147);
110 FOR X = 1 TO 23
120 FOR Y = 1 TO 22
130 IF X = 23 AND Y = 22 THEN PRINT CHR$(157) CHR$(148);
140 PRINT "Z";
150 NEXT: NEXT
160 GOTO 160
```

This program will fill the entire screen with Z's, including the bottom right corner, and then freeze. The IF in line 120 checks for both X=23 and Y=22 being true, or else the program just drops through to line 130. When the conditions are true, the VIC PRINTs a cursor left and an insert.

By the way, this really is a trick to PRINT in the lower right corner in the screen without forcing the screen to scroll up a line. This is because you never really PRINT in that position, you insert in the position before that, which pushes the character into position.

For those of you using cartridges that add extra commands to BASIC, like "Super Expander" and "Programmers Aid," make sure that you put a colon between the word THEN and one of the extra commands.

EXAMPLE 4:

```
100 IF X=4 THEN : GRAPHIC 4
```

INPUT

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
INPUT [<u>"prompt";</u> <u>variable</u>]	None	None

This is a statement that lets the person running the program "feed" information into the computer. When executed, this statement PRINTs a question mark (?) on the screen, and positions the cursor 2 spaces to the right of the question mark. Now the computer waits, cursor blinking, for the operator to type in the answer and press the RETURN key.

The word INPUT may be followed by any text contained in quote marks (""). This text is PRINTed on the screen, followed by the question mark.

After the text comes the name of one or more *variables* separated by commas. This variable is where the computer stores the information that the operator types. The variable can be any legal variable name, and you can have several different variable names, each for a different input.

EXAMPLE 1:

```
100 INPUT A
110 INPUT B, C, D
120 INPUT "PROMPT"; E
```

When this program runs, the question mark appears to prompt the operator that the VIC is expecting an input for line 100. Any number typed in goes into A, for later use in the program. If the answer typed was not a number, the ?REDO FROM START message appears, which means that a string was received when a number was expected. If the operator just hit RETURN without typing anything, the variable's value doesn't change.

Now the next question mark, for line 110, appears. If we type only one number and hit RETURN, the VIC will now display 2 question marks (??), which means that more input is required. You can just type as many inputs as you need separated by commas, which prevents the double question mark from appearing. If you type more data than the INPUT statement requested, the ?EXTRA IGNORED message appears, which means that the extra items you typed were not put into any variables.

Line 120 displays the word PROMPT before the question mark appears. The semicolon is required between the prompt and any list of variables. Note: The only way to end a program during an INPUT statement is to hold down the RUN/STOP key and hit RESTORE.

EXAMPLE 2:


```
10 PRINT "INPUT A WORD":INPUT A$
20 PRINT "YOUR INPUT WAS" A$
30 GOTO 10
```

LET

Format:

[LET] variable =
expression

Abbreviation:

L  E

Screen Display:

L 

The LET statement is used to set a variable to a value. The value can be a constant (like 5) another variable (like C), or a complex formula (like $PI * R + 3$). The LET statement also works with string variables.

Since the LET statement is used so often in BASIC programs, the word 'LET' has been made optional, since it did nothing but take up memory. Advanced programmers always leave it out.

```
LET B = 1
A = 3 * 3 + 33 - B
A$ = "CAT" + "DOG"
```

This is the same as B = 1
A will equal 41
A\$ will equal "CATDOG"

NEXT

Format:

NEXT [variable , variable
 , . . . , variable]

Abbreviation:

N **SHIFT** E

Screen Display:

N 

This statement completes a loop that was started by a FOR statement. If the word NEXT is not followed by a variable name, the loop completed is the last one that was started. If there is a variable name given, that loop is finished. If the loop being finished wasn't the last one started, any loops between the last one and the one specified in the NEXT statement are lost.

Care must be taken when placing loops within other loops, to complete them properly. Here is an example of correct placement ("nesting") of loops.

EXAMPLE 1:

```
10 FOR L = 1 TO 100
20 FOR M = 1 TO 10
30 NEXT M
40 NEXT L
```

Notice that the first loop finished is the last one that was started. Here are some general examples of the NEXT statement.

EXAMPLE 2:

```
NEXT
NEXT J
NEXT I, J, K
```

ON

Formats:

ON variable GOTO number [, number , . . . , number]
ON variable GOSUB number [, number , . . . , number]

Abbreviation:

Screen Display:

This statement allows the program to choose from a list of line numbers to go to. If the variable has a value of 1, the first line number is the one chosen. If the value is 2, the second number in the list is used, and so on. If the value in the variable is less than 1 or greater than the number of line numbers in the list, the program just

ignores the statement and continues with the statement following the ON statement.

EXAMPLE 1:

```
ON X GOTO 100, 130, 180, 220
ON X+3 GOSUB 9000,20,9000
```

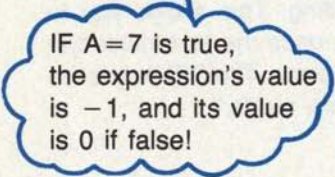
ON is really an under-used variant of the IF ... THEN ... statement, which can send the program to one of many possible lines. Using formulas and logical operators, one ON statement can replace a whole list of IF statements.

EXAMPLE 2: IF statements

```
IF A=7 THEN 400
IF A=3 THEN 900
IF A < 3 THEN 1000
IF A > 7 THEN 100
```

EXAMPLE 3: ON ... GOTO ...

```
ON -(A=7) - 2*(A=3) - 3*(A<3) - 4*(A>7) GOTO 400, 900,
1000, 100
```



IF A=7 is true,
the expression's value
is -1, and its value
is 0 if false!

POKE

Format:

POKE location, value

Abbreviation:

P **SHIFT** O

Screen Display:

P ☐

This statement allows you to alter the value of any RAM location in memory. There are a possible 65,536 locations in the VIC's memory, and in an unexpanded VIC a little more than 5K of them are RAM and can be changed. Your 5K of RAM is in locations numbered from 0 to 1023 and from 4096 to 8191. The memory maps in Chapter 3 describe the contents of the first 1K. Your BASIC program, variables, and the screen memory all go in the 4K area.

Color RAM is an extra half-K block of memory starting at 38400. There are also alterable areas in some of the chips, like the VIC chip from 36864 to 36879 and the 6522 chips above that.

Because each memory location holds 1 byte, which can have a value from 0 to 255, only numbers in that range can be POKEd into memory.

EXAMPLE:

POKE 36879 , 8

POKE A , B

PRINT

Abbreviation: ?















There is no statement in BASIC with more variety than the PRINT statement. There are so many symbols, functions, and parameters associated with this statement that it might almost be considered as a language of its own within BASIC, a language specially designed for writing on the screen.

Quote mode






Once the quote mark (SHIFT 2) is typed, the cursor controls stop operating and start displaying reversed characters which actually stand for the cursor control you are hitting. This allows you to program these cursor controls, because once the text inside the quotes is PRINTed they perform their functions. The DEL key is the only cursor control not affected by "quote mode."

1. Cursor movement






The cursor controls which can be "programmed" in quote mode are:

Key	Appears as
	
	
	
	
	
	
	

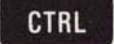





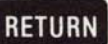
If you wanted the word HELLO to PRINT diagonally from the upper left corner of the screen, you would type:

PRINT "  H  E  L  L  O"

or

















PRINT "  H  E  L  L  O"

2. Reverse characters

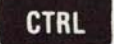

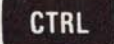

Holding down the  key and hitting  will cause a  to appear inside the quotes. This will make all characters start printing in reverse video (like a negative of a picture). To end the reverse printing, hit  , which prints a  or else PRINT a . (Just ending the PRINT statement without a semicolon or comma will take care of this.)

3. Color controls



Holding down the CTRL key with any of the 8 color keys will make a special reversed character appear in the quotes. When the character is PRINTed, then the color change will occur.

Color	Appears as
	
	
	
	
	
	
	
	






If you wanted to PRINT the word HELLO in cyan and the word THERE in green, type:

PRINT "   HELLO   THERE"

or

PRINT "  HELLO  THERE"



4. Insert mode

The spaces created by using the insert  key have some of the same characteristics as quote mode. The cursor controls and color controls show up as reversed characters. The only difference is in the  and , which performs its normal function even in quote mode, now creates the . And , which created a special character in quote mode, inserts spaces normally.

Because of this, it is possible to create a PRINT statement containing DEletes, which cannot be PRINTed in quote mode. Here is an example of how this is done:

```
10 PRINT"HELLO"      P"
```

which displays as

```
10 PRINT"HELLO"   P"
```

When the above line is RUN, the word displayed will be HELP, because the last two letters are deleted and the P is put in their place.

WARNING: The DEletes will work when LISTing as well as PRINTing, so editing a line with these characters will be difficult.

The "insert mode" condition is ended when the RETURN (or SHIFT RETURN) key is hit, or when as many characters have been typed as spaces were inserted.

5. Other special characters

There are some other characters that can be PRINTed for special functions, although they are not easily available from the keyboard. In order to get these into quotes, you must leave empty spaces for them in the line, hit RETURN or SHIFT RETURN, and go back to the spaces with the cursor controls. Now you must hit CTRL RVS ON, to start typing reversed characters, and type the keys shown below:

Function	Type
SHIFT RETURN	SHIFT M
switch to lower case	N
switch to upper case	SHIFT N
disable case-switching keys	H
enable case-switching keys	I

The SHIFT RETURN will work in the LISTing as well as PRINTing, so editing will be almost impossible if this character is used. The LISTing will also look very strange.

READ

Format:

READ variable list

Abbreviation:

R **SHIFT** E

Screen Display:

R ☐

This works with the DATA statement to fill variables with values stored within the program. The information is usually in the form of a list that is READ in at the beginning of the program, or a table that is re-READ during the program. READ works just like INPUT, except that the information comes from DATA statements instead of the person working the program.

EXAMPLE:

```
10 READ A, B, C$  
20 DATA 1, 2, HELLO THERE!
```

REM

Format:

REM any text

Abbreviation:

None

Screen Display:

None

This statement makes your program more easily understood when LISTed. It is a reminder to yourself to tell you what you had in mind when writing each section. For instance, you might tell what a variable is used for, what date the program was written, or some other useful piece of information. The REMark can be any text, word, or character, including the colon (:) or BASIC keywords. Therefore, the REM statement is the last one on a line that the program sees.

If you try to use graphic characters in a REM statement without using a quote mark (") first, when you LIST the line you'll see BASIC keywords instead of the graphic characters. This is because the VIC thinks these characters are the "tokens" for those commands. The BASIC tokens are discussed in the last part of this chapter.

EXAMPLE:

REM PROGRAM BY SUE M. 10/6/81	Good example
REM A\$ HOLDS 22 CURSOR DOWNS	Good example
LET A = 1 : REM PUT A 1 IN A	Bad example

RESTORE

Format:

RESTORE

Abbreviation:

RE **SHIFT** S

Screen Display:

RE ☒

This statement sets the DATA statement pointer back to the first DATA statement in the program. Each time you READ the DATA, the pointer advances through all the items in the first DATA statement, then through the items in the next DATA statement, and so on through all the DATA statements in the program. In order to re-READ the items, use the RESTORE statement.

EXAMPLE:

```
10 DATA 1, 2, 3, 4
20 DATA 5, 6, 7, 8
30 FOR L = 1 TO 8
40 READ A : PRINT A
50 NEXT
60 RESTORE
70 FOR L=1 TO 8
80 READ A : PRINT A
90 NEXT
```

RETURN

Format:

RETURN

Abbreviation:

RE **SHIFT** T

Screen Display:

RE ☐

This statement completes a subroutine that was begun with the GOSUB statement. When the GOSUB is performed, the VIC remembers which line it came from. When it later hits a RETURN statement, it goes back to the statement right after the original GOSUB. This is similar to a GOTO, except the GOSUB subroutine can be performed and the program *continued* from the original GOSUB line.

EXAMPLE:



```
10 PRINT "THIS IS THE PROGRAM"
20 GOSUB 1000
30 PRINT "PROGRAM CONTINUES"
```


```

40 GOSUB 1000
50 PRINT "MORE PROGRAM"
60 END
1000 PRINT "THIS IS THE GOSUB": RETURN

```

STOP

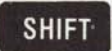

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
STOP	S  T	S 

This statement will halt a program and return control to the user. The only difference between the STOP and END statements is that the message BREAK IN LINE XXXX appears when STOP is used, just as if the user had pressed the  key.

EXAMPLE:

```
100 STOP
```

SYS

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
SYS location	S  Y	S 

This is the most common way to mix a BASIC program with a machine language program. The machine language program begins at the location given in the SYS statement. When the machine language instruction RTS (return from subroutine) is reached, the program jumps back to the BASIC program, right after the SYS statement. A machine language program can be POKEd into memory from BASIC or created with the aid of VICMON™.

EXAMPLE:

```

SYS 64802           resets the VIC from power-up
POKE 4400 , 96 : SYS 4400  returns immediately

```

WAIT

Format:

WAIT location,
mask1[, mask2]

Abbreviation:

W **SHIFT** A

Screen Display:

W 

For most programmers, this statement should never be used. It causes the program to halt until a specific memory location's bits change in a specified way. This is used for arcane I/O operations and almost nothing else.

The WAIT statement takes the value in the memory location and performs a logical AND operation with the value in mask1. If there is a mask2 in the statement, the result of the first operation is exclusive-ORed with mask2. This sounds confusing, but there's an easier way to look at it. The mask1 value "filters out" any bits that we don't want to test. Where the bit is 0 in mask1, the corresponding bit in the result will always be 0. The mask2 value will flip any bits, so we can test for an off condition as well as on. Any bits being tested for a 0 should have a 1 in the corresponding position in mask2.

EXAMPLE:

WAIT 36868 , 144 , 16

What are we testing for here? Here's a binary look at our two masks:

144 = 10010000

16 = 00010000

This WAIT statement will halt the program until either the 128 bit is on or the 32 bit is off.

I/O STATEMENTS

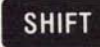
CLOSE
CMD
GET#
INPUT#
OPEN
PRINT#

CLOSE

Format:

CLOSE file#

Abbreviation:

CL  **O**

Screen Display:

CL 

This closes the file that was started in an OPEN statement. It is recommended that a PRINT# to that file be performed before closing the file, to make sure that all data has been transmitted. Not closing an OPEN file results in a FILE OPEN ERROR.

EXAMPLE:

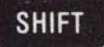
OPEN 1,4 : PRINT#1 , "HI THERE!" : CLOSE 1

CMD

Format:

CMD file#

Abbreviation:

C  **M**

Screen Display:

C 

This changes the normal output device of the VIC from the screen to the file specified. In this way, data and LISTings can be sent to other devices, like the printer, disk, or tape drive. When finished transmitting, to reset output to the screen, do a PRINT# and CLOSE the file.

EXAMPLE:

OPEN 1,4 : CMD 1 : PRINT "HELLO THERE!" : PRINT#1 : CLOSE 1

GET#

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
GET# <u>file#</u> , <u>variable</u>	None	None



This statement receives data one byte at a time from any OPENED device. If no data is available, it works the same as the GET statement, returning a null value. The INPUT# statement can get more than one character, and will get all the characters up to a carriage return (CHR\$(13)). The GET# will receive any characters, 1 at a time, including special characters like the carriage return and quote marks.

EXAMPLE:

```
10 OPEN 1, 3
20 PRINT CHR$(147) "HELLO THERE" CHR$(19);
30 FOR L = 1 TO 22
40 GET#1, B$: A$ = A$ + B$
50 NEXT : PRINT A$ : CLOSE 1
```

If you examine A\$ when this program is finished, you'll see that the last character is a CHR\$(13), a carriage return, which terminates the line.

INPUT#

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
INPUT# <u>file#</u> , <u>variable 1[, variable 2, etc.]</u>	I  N	I 

This usually is the fastest and easiest way to retrieve data that was stored in a file on tape or disk. The data is in the form of whole variables, as opposed to the one-byte-at-a-time method of GET#. First the file must have been OPENed, then you can use INPUT# to fill your variables.

EXAMPLE:

```
10 OPEN1, 1, 0, "TAPE FILE NAME"
20 PRINT "FILE IS OPEN OK"
30 INPUT# 1, A$, B$
40 CLOSE 1
```

When using the screen (device # 3) as an input device, the INPUT# statement can be used to read any whole line of text from the screen. The last character of the line will be read as a CHR\$(13), as if the screen hit the RETURN key at the end of the line!

However, there are times when it's not always practical to use INPUT#, and some precautions are in order. If a string variable put on the file has certain characters, using INPUT# could have unexpected results. If you use CHR\$(13), or a comma (,) or semicolon (;) or colon (:), the VIC will think that this marks the end of your variable. If you put quote marks (CHR\$(34)) at the start and end of your string when it is written, it will come back intact.

INPUT # may also be used to "INPUT" data without the question mark (?) prompt being displayed. This is very useful for a variety of applications, for example if you want to set up a graphic chart and let the operator INPUT data to the chart without question marks being displayed.

EXAMPLE:

```
10 OPEN 1,0
20 PRINT "ENTER A NUMBER": INPUT#1, A
30 PRINT A "TIMES" 5 "EQUALS" A*5
```

OPEN

Format:

OPEN file#,

Abbreviation:

O **SHIFT** P

Screen Display:

O ☐

[device#, command#, string]

This statement OPENS a channel for input and/or output to a device. This device can be part of the VIC, like the screen and keyboard, or an accessory, like the tape recorder, printer, or disk drive. When OPENing a channel to an external device, the VIC sets up a buffer for the data, and only transmits and receives whole buffers at a time.

The file# can be any number from 1 to 255, and is the same number that will be used in the INPUT#, GET#, and PRINT# statements to work with this device. The device# specifies which device, and is set within that device.

DEVICE#	DEVICE
0	keyboard
1	cassette deck
2	RS232 device
3	screen
4	printer
5	printer
8	disk drive
4-127	serial bus device
128-255	serial bus device—send lf after cr

The command# is specific to each different device. Here are some of the command numbers:

DEVICE	COMMAND#	EFFECT
Cassette	0	read tape file
	1	write tape file
	2	write tape file, put EOT marker at end
Disk	1-14	open data channel
	15	open command channel
Keyboard	1-255	no effect
Screen	1-255	no effect
Printer	0	upper case/graphics
	7	upper/lower case
RS232		See RS232 (Section 4)

The string at the end is sent to the printer or screen as if a PRINT# were performed to that device. With the cassette deck, it is used for the filename, and with disk it can be a filename or some control information.

EXAMPLE:

OPEN 1, 0	Read the keyboard
OPEN 1, 1, 0, "name"	Read from cassette

OPEN 1, 1, 1, "name"	Write to cassette
OPEN 1, 1, 2, "name"	Write to tape, put EOT marker after file
OPEN 1, 2, 0, "string"	Open channel to RS232 device
OPEN 1, 3	Read/write screen
OPEN 1, 4, 0, "string"	Send upper case/graphics to printer
OPEN 1, 4, 7, "string"	Send upper/lower case to printer
OPEN 1, 5, 0, "string"	Send upper/lower case to printer, device# switched
OPEN 1, 8, 15, "command"	Send command to disk

PRINT#

Format:

PRINT# file#,
variable list

Abbreviation:

P **SHIFT** R

Screen Display:

P ☐

This sends the contents of the variables in the list to the device that was previously OPENed. They will be transmitted in the same format as if PRINTed to the screen; if commas are used as separators extra spaces will appear, if semicolons are used no space will appear, and a CHR\$(13) is the last character sent if there isn't a comma or semicolon at the end of the line. The easiest way to write more than one variable to a file on tape or disk is to set a string variable to CHR\$(13), and use that string in between all the other variables when writing the file.

EXAMPLE:

```
100 OPEN 1, 1, 1, "TAPE FILE"
110 R$ = CHR$(13)
120 PRINT#1, 1; R$; 2; R$; 3; R$; 4; R$; 5
130 PRINT#1, 6
140 PRINT#1, 7
```

The example shows how to write a tape file that can be easily read back using INPUT# statements, since each variable has a CHR\$(13) printed after it. You can also print "," or ";" to separate the variables.

VIC 20 BASIC FUNCTIONS

The intrinsic functions provided by BASIC are presented in the following paragraphs. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

- X and Y Represent any numeric expression
- I and J Represent integer expressions
- X\$ and Y\$ Represent string expressions

If a floating point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

VIC 20 BASIC Functions

FUNCTION	RESULT	
	Numeric	String
ABS	X	
ASC	X	
ATN	X	
CHR\$		X
COS	X	
EXP	X	
FRE	X	
INT	X	
LEFT\$		X
LEN	X	
LOG	X	
MID\$		X
PEEK	X	
POS	X	
RIGHT\$		X
RND	X	

SGN	X
SIN	X
SPC	X
SQR	X
STATUS	X
STR\$	X
TAB	X
TAN	X
TIME	X
TIME\$	X
USR	X
VAL	X

ABS

Format:

ABS(X)

Abbreviation:

A **SHIFT** B

Screen Display:

A 

Action: Returns the absolute value of the expression X.

EXAMPLE:

PRINT ABS (7*(-5))

35

READY.

ASC

Format:

ASC(X\$)

Abbreviation:

A **SHIFT** S

Screen Display:

A 

Action: Returns a numerical value that is the ASCII code of the first character of the string X\$. (See Appendix F for ASCII codes.) If X\$ is null, an "ILLEGAL QUANTITY" error is returned.

EXAMPLE:

```
10 X$ = "TEST"  
20 PRINT ASC(X$)  
RUN  
84  
READY.
```

See the CHR\$ function for ASCII-to-string conversion.

ATN

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
ATN(X)	A SHIFT T	A <input type="checkbox"/>

Action: Returns the arctangent of X in radians. Result is in the range $-\pi/2$ to $\pi/2$. The expression X may be any numeric type, but the evaluation of ATN is always performed in floating point binary.

EXAMPLE:

```
10 INPUT X  
20 PRINT ATN(X)  
RUN  
? 3  
1.24904577  
READY.
```

CHR\$

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
CHR\$(I)	C SHIFT H	C <input type="checkbox"/>

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix F.) CHR\$ is commonly used to send a special character to the terminal. For instance, a screen clear could be sent (CHR\$(147)) to clear the CRT screen and return the cursor to the home position, as a preface to an error message.

EXAMPLE:

```
PRINT CHR$(66)  
B  
READY.
```

See the ASC function for ASCII-to-numeric conversion.

COS



<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
COS(X)	None	None

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in floating point binary.

EXAMPLE:

```
10 X=2*COS(.4)
20 PRINT X
RUN
  1.84212199
READY.
```

EXP



<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
EXP(X)	E  X	E 

Action: Returns e to the power of X. X must be ≤ 88.02969191 . If EXP overflows, the "OVERFLOW" error message is displayed.

EXAMPLE:

```
10 X=5
20 PRINT EXP (X-1)
RUN
  54.5981501
READY.
```

FRE

<u>Format:</u>	<u>Abbreviation:</u>	<u>Screen Display:</u>
FRE(X)	F  R	F 

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC.

EXAMPLE:

```
PRINT FRE(0)
14542
READY.
```

INT**Format:**

INT(X)

Abbreviation:

None

Screen Display:

None

Action: Returns the largest integer $\leq X$.

EXAMPLE:

```
PRINT INT(99.4343), INT(-12.34)
99      -13
Ready.
```

LEFT\$**Format:**

LEFT\$(X\$, I)

Abbreviation:LE **SHIFT** F**Screen Display:**LE

Action: Returns a string comprised of the leftmost I characters of X\$. I must be in the range 0 to 255. If I is greater than LEN(X\$), the entire string (X\$) will be returned. If I = 0, the null string (length zero) is returned.

EXAMPLE:

```
10 A$ = "COMMODORE COMPUTER"
20 B$ = LEFT$(A$, 9)
30 PRINT B$
COMMODORE
READY.
```

Also see the MID\$ and RIGHT\$ functions.

LEN

Format:

LEN(X\$)

Abbreviation:

None

Screen Display:

None

Action: Returns the number of characters in X\$. Non-printing characters and blanks are counted.

EXAMPLE:

```
10 X$ = "COMMODORE COMPUTER"  
20 PRINT LEN (X$)  
18  
READY.
```

LOG

Format:

LOG(X)

Abbreviation:

None

Screen Display:

None

Action: Returns the natural logarithm of X. X must be greater than zero.

EXAMPLE:

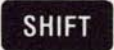
```
PRINT LOG (45/7)  
1.86075234  
READY.
```

MID\$

Format:

MID\$(X\$, I[,J])

Abbreviation:

M  I

Screen Display:

M 

Action: Returns a string of length J characters from X\$ beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I > LEN(X\$), MID\$ returns a null string.

EXAMPLE:

LIST

10 A\$="GOOD"

20 B\$="MORNING EVENING AFTERNOON"

30 PRINT A\$;MID\$(B\$,9,7)

RUN

GOOD EVENING

READY.


Also see the LEFT\$ and RIGHT\$ functions.

PEEK

Format:

PEEK(I)

Abbreviation:

P  E

Screen Display:

P 

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65535. PEEK is the complementary function to the POKE statement.

EXAMPLE:

PRINT PEEK(36879)

This will return the value of the screen background color byte.

POS

Format:

POS(X)

Abbreviation:

None

Screen Display:

None

Action: Returns the current cursor position. The leftmost position is 0. X is a dummy argument.

EXAMPLE:

IF POS(X) >20 THEN PRINT CHR\$(13)

RIGHT\$

Format:

RIGHT\$(X\$, I)

Abbreviation:

R **SHIFT** I

Screen Display:

R 

Action: Returns the rightmost I characters of string X\$. If I=LEN(X\$), returns X\$. If I=0, the null string (length zero) is returned.

EXAMPLE:

```
10 A$="COMMODORE BUSINESS MACHINES"  
20 PRINT RIGHT$(A$, 8)  
RUN  
MACHINES  
READY.
```

Also see the MID\$ and LEFT\$ functions.

RND

Format:

RND(X)

Abbreviation:

R **SHIFT** N

Screen Display:

R 

Action: Returns a random number between 0 and 1. X>0 returns the same pseudo-random number sequence for any random number seed. X<0 reseeds the sequence, each X producing a different seed. The sequence is seeded at random on power-up. X=0 generates a random number from a free running clock.

EXAMPLE:

```
10 FOR I=1 TO 5  
20 PRINT INT(RND(X)*100);  
30 NEXT  
RUN  
24 30 31 51 5  
READY.
```

SGN

Format:

 $\text{SGN}(X)$

Abbreviation:

S **SHIFT** G

Screen Display:

S ☐

Action: IF $X > 0$, $\text{SGN}(X)$ returns 1. If $X = 0$, $\text{SGN}(X)$ returns 0. If $X < 0$, $\text{SGN}(X)$ returns -1.

EXAMPLE:

[illegible]

SIN

Format:

SIN(X)

Abbreviation:

S SHIFT I

Screen Display:

S ☐

Action: Returns the sine of X in radians. SIN(X) is calculated in floating point binary. COS(X)=SIN(X+3.14159265/2)

EXAMPLE:

PRINT SIN (1.5)
.997494987
READY.

SPC

Format:

SPC(I)

Abbreviation:

S SHIFT P

Screen Display:

S ☐

Action: Prints I blanks on the screen. SPC may only be used with PRINT. I must be in the range 0 to 255.

EXAMPLE:

```
PRINT "OVER" SPC(15) "THERE"  
OVER          THERE  
READY
```

SQR

Format:

SQR(X)

Abbreviation:

S **SHIFT** Q

Screen Display:

S 

Action: Returns the square root of X. X must be ≥ 0 .

EXAMPLE:

```
10 FOR X = 10 TO 25 STEP 5
```

```
20 PRINT X, SQR(X)
```

```
30 NEXT
```

```
RUN
```

```
10          3.16227766
```

```
15          3.87298335
```

```
20          4.47213595
```

```
25          5
```

```
READY.
```

STATUS

Format:

STATUS

Abbreviation:

ST

Screen Display:

ST

Action: Returns the CBM status corresponding to the last I/O operation, whether over cassette, screen, keyboard or serial bus.

ST bit position	ST numeric value	Cassette Read	Serial Bus R/W	Tape verify + load
0	1		time out write	
1	2		time out read	
2	4	short block		short block
3	8	long block		long block
4	16	unrecoverable read error		any mismatch

ST bit position	ST numeric value	Cassette Read	Serial Bus R/W	Tape verify + load
5	32	checksum error		checksum error
6	64	end of file	EOI	
7	-128	end of tape	device not present	end of tape

EXAMPLE:

```

10 OPEN 2,1,2, "MASTER FILE"
12 GET#2,A$
14 IF STATUS AND 64 THEN 20
16 ?A$
18 GOTO12
20 ?A$: CLOSE2

```

STR\$

Format:

STR\$(X)

Abbreviation:

ST **SHIFT** R

Screen Display:

ST ☐

Action: Returns a string representation of value of X.

EXAMPLE:

```

5 REM LINE UP DECIMAL POINTS
10 INPUT "TYPE A NUMBER";N
20 A$ = STR$(N): Q = LEN(A$)
30 FOR L = Q TO 1 STEP -1
40 IF MID$(A$, L, 1) <> "." THEN NEXT: A$ = A$ + ".00":
   GOTO 60
50 IF L = Q - 1 THEN A$ = A$ + "0"
60 PRINT TAB(10)A$
70 GOTO 10
Also see the VAL function.

```

TAB

Format:

TAB(I)

Abbreviation:

T **SHIFT** A

Screen Display:

T 

Action: Spaces to position I on the screen. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 0 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 0 to 255. TAB may only be used in PRINT.

EXAMPLE:

```
10 PRINT "NAME" TAB(15) "AMOUNT" : PRINT
20 READ A$, B$
30 PRINT A$ TAB(15)B$
40 DATA "G.T. JONES", "$25.00"
RUN
NAME           AMOUNT
G. T. JONES     $25.00
READY
```

TAN

Format:

TAN(X)

Abbreviation:

None

Screen Display:

None

Action: Returns the tangent of X in radians. TAN(X) is calculated in binary. If TAN overflows, the "OVERFLOW" error message is displayed.

EXAMPLE:

```
10 Y = Q*TAN(X)/2
```

TIME

Format:

TI

Abbreviation:

None

Screen Display:

None

Action: Used to read the internal interval timer and return a value in one-tenth seconds. This is a real-time clock. This value is initialized only when TI\$ is invoked.

EXAMPLE:

```
10 PRINT TI/60 "SECONDS SINCE POWER UP"
```

TIMES

Format:

TI\$

Abbreviation:

None

Screen Display:

None

Action: Used to read the internal interval timer and return a string of 6 characters in hours, minutes, seconds. May be used in an input statement or on the left hand side of an expression to initialize the timer.

EXAMPLE:

```
10 TI$ = "000000"  
20 FOR I=1 TO 10000:NEXT  
30 PRINT TI$  
RUN  
000010  
READY.
```

USR

Format:

USR(X)

Abbreviation:

U  S

Screen Display:

U 

Action: Calls the user's assembly language subroutine whose starting address is stored in locations 1 and 2. The argument is stored in the floating point accumulator (see memory map), and the result is the value residing there when the routine returns to BASIC.

EXAMPLE:

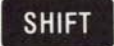
```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
etc.
```

VAL

Format:

VAL(X\$)

Abbreviation:

V  A

Screen Display:

V 

Action: Returns the numerical value of string X\$. If the first character of X\$ is not +, -, \$, or a digit, VAL(X\$)=0

EXAMPLE:

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL (ZIP$) < 90000 OR VAL (ZIP$) > 96699 THEN PRINT
   NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
   PRINT NAME$ TAB(25) "LONG BEACH"
40 DATA "SUE M.", "MEDIA", "PA", "19063"
(See the STR$ function for numeric to string conversion.)
```

NUMBERS AND VARIABLES

The numbers printed by VIC 20 are governed by limitations within their formats. The numbers are stored internally to over ten digits of accuracy. However, when a number is printed, only nine digits are displayed. Each number may also have an exponent (a power-of-ten scaling factor).

Numbers are used all the time when working with VIC 20. There are two kinds of numbers that can be stored: floating point numbers (also called real numbers) and integers.

Floating point is the standard number representation used by the VIC. The VIC does its arithmetic using floating point numbers. A floating point number can be a whole number, a fractional number preceded by a decimal point, or a combination. The number can be signed negative (-) or positive (+). If the number has no sign, it is assumed to be positive.

Consider the following examples of floating point numbers:

Whole number equivalent to an integer:

5

- 15

65000

161

0

Numbers with a decimal point:

0.5

0.0165432

- 0.00009

1.6

24.0085

- 65.6

3.1416

Notice that if you put a comma in a number and ask the VIC to assign it to a variable, you will get a Syntax Error message. For example, you must use 65000, not 65,000.

Numbers always have at least eight digits of precision; they can have up to nine, depending on the number. The VIC rounds any additional significant digits. It rounds up when the next digit is five or more; it rounds down when the next digit is four or less.

Rounding numbers will sometimes cause fractional numbers to look inaccurate. Here are some examples:

You type: ? .5555555556
VIC prints: .55555555
(VIC appears to round down on 6 or less; up on 7 or more.)

You type: \$.555555557
VIC prints: .55555556

You type: ? .1111111115
VIC prints: .11111111
(VIC appears to round down on 5 or less; up on 6 or more).

You type: ? .1111111116
VIC prints: .11111112

These quirks result from the manner in which computers store floating point numbers.

Floating point numbers can also be represented in scientific notation. When numbers with ten or more digits are entered, the VIC automatically converts them to scientific notation. Scientific notation allows the VIC to accurately display these large numbers using fewer digits. For example:

READY.
?1111111114
1.11111111E+09

READY.
?1111111115
1.11111112E+09

A number in scientific notation has the form:

numberE ± ee

Where:

- | | |
|--------|--|
| number | is an integer, fraction, or combination, as illustrated above. The "number" portion contains the number's significant digits; it is called the "coefficient." If no decimal point appears, it is assumed to be to the right of the coefficient. |
| E | is the upper case letter E. |
| ± | is an optional plus sign or minus sign which indicates the sign of the exponent. |
| ee | is a one- or two-digit exponent. The exponent specifies the magnitude of the number; that is, the number of places to the right (positive exponent) or to the left (negative exponent) that the decimal point must be moved to give the true decimal point location. |

Here are some examples:

Scientific Notation

2E1
10.5E + 4
66E + 2
66E - 2
- 66E - 2
1E - 10
94E20

Standard Notation

20
105000
6600
0.66
- 0.66
0.0000000001
9400000000000000000000

As the last two examples show, scientific notation is a much more convenient way of expressing very large or very small numbers. VIC BASIC prints numbers ranging between 0.01 and 999,999,999 using standard notation; however, numbers outside of this range are printed using scientific notation.

Consider the following out-of-range examples:

? .009
9E-03

READY.
?.01
.01

READY.
?999999998.9
999999999

READY.
?999999999.6
1E + 09

There is a limit to the magnitude of a number that the VIC can handle, even in scientific notation. The range limits are:

Largest floating point number: $\pm 1.70141183E + 38$
Smallest floating point number: $\pm 2.93873588E - 39$

Any number of larger magnitude will give an overflow error. For example:

?1.70141184E + 38
?OVERFLOW ERROR
READY.

Any number of a smaller magnitude will yield a zero result. For example:

?2.93873587E - 39
0

READY.

An integer is a number that has no fractional portion or decimal point. The number can be signed negative (-) or positive (+). An unsigned number is assumed to be positive. Integer numbers have a limited range of values, from -32768 to +32767.

The following are examples of integers:

0
1
44
32699
-15

Any number that is an integer can also be represented in floating point format since integers are a subset of floating point numbers. VIC BASIC converts any integers to floating point representation before doing arithmetic with them. The most important difference between floating point numbers and integers is that an integer array uses less storage space in memory (two bytes for an integer, versus five bytes for a floating point number).

We have already used strings as messages to be printed on the display screen. A string consists of one or more characters enclosed in double quotation marks.

Consider the following examples of strings:

"HI"
"SYNERGY"
"12345"
"\$10.89 IS THE AMOUNT"

All of the data keys (alphabetic, numeric, special symbols, and graphics), the three cursor control keys (Clear Screen/Home, Cursor Up/Down, Cursor Left/Right), as well as the Reverse On/Off key, Insert/Delete, and Stop keys can be included in a string. The only keys that cannot be used within a string are Return, CTRL, Shift, and the Logo key.

All characters within the string are displayed as they appear. The cursor control and Reverse On/Off keys, however, normally do not print anything themselves; to show that they are present in a string, certain reverse field symbols are used. They are shown in Table 2-1.

When you enter a string from the keyboard, it can have any length up to the space available within an 88-character line (that is, any character spaces not taken up by the line number and other required parts of the statement). However, strings of up to 255 characters can be stored in the VIC's memory. You get long strings

by pushing together, or concatenating, two separate strings to form one longer string. We will describe this further when we discuss string variables in general.

Earlier in the chapter, we introduced the concept of a variable. In this discussion variables are described more thoroughly.

A variable is a data item whose value may be changed. The value is determined by the number assigned to the variable. If you type the immediate mode statement:

```
PRINT 10, 20, 30
10      20      30
```

The VIC will display the same three numbers (as illustrated above) whenever the PRINT statement is executed; that is because this PRINT statement used constant data. However, you can write the immediate mode statement:

```
A = 10: B = 20: C = 30: PRINT A, B, C
10      20      30
```

The same three numbers, 10, 20, 30, are displayed; however, A, B, and C are variables, not constants. By changing the values assigned to A, B, and C, you can change the values printed out by the PRINT statement. Consider the following example of this:

```
A = -4: B = 45: C = 4E2: PRINT A, B, C
-4      45      400
```

You will notice that variables appear in virtually all computer programs.

Variables are identified by names. We used A, B, and C as variable names in the illustrations above. A variable has two parts: its name and a value. The variable name represents a location at which the current value is stored. In the following illustration, the current value of A is 14; for B it is 15; and for C it is 0.

Variable Name	Contents
A	14
B	15
C	0

If we change A to -1 using the immediate mode statement:

```
A = -1
```

then the Location Contents, stored under the variable-name A, will change from 14 to -1.

This is an excellent way of looking at variables because it is, in

fact, the way they are handled by the VIC. A variable name represents an address in memory; and at that memory location, the current value of the variable is stored. The important point to note is that variable names—which are names that programmers make up—are arbitrary; they have no innate relationship to the value that the variables represent.

A variable name can have one or two characters. The first character must be an alphabetic character from A to Z; the second character can be either alphabetic or numeric (numeric characters must be in the range from 0 to 9). A third character can be included to indicate the type of number (integer or string), % or \$.

Floating variables represent floating point numbers. This is probably the most common type of variable that you will use.

The following are examples of floating point variables:

A
B
A1
AA
Z5

Integer variables represent integers. Names for integer variables are followed by the % symbol as the following examples indicate:

A%
B%
A1%
MN%
X4%

Remember, floating point variables can also represent integers; but you should use integer variables in arrays whenever possible since they use less memory—two bytes versus five for a floating point array element.

A string variable represents a string of text. The following are examples of string variables:

A\$
M\$
MN\$
M1\$
ZX\$
F6\$

You can use variable names having more than two alphanumeric characters; but if you do, only the first two characters count. To VIC

BASIC, therefore, BANANA and BANDAGE are the same name since both begin with BA.

The advantage of using longer variable names is that they make programs easier to read. PARTNO, for example, is more meaningful than PA as a variable name describing a part number in an inventory program.

VIC BASIC allows variable names to have up to 255 characters. The following are examples of names with more than the minimum number of characters:

```
MEMBERS$  
T1234567  
BBB$  
ABCDPG%  
PARTY
```

If you use extended variable names, keep in mind the following:

1. Only the first two characters, plus the identifier symbol, are significant in identifying a variable. Do not use extended names like LOOP1 and LOOP2; these refer to the same variable: LO.
2. VIC BASIC has what are called "reserved words." These are words that have special meaning for the VIC BASIC interpreter. No variable can contain a reserved word. In longer names you have to be very careful that a reserved word does not occur embedded anywhere in the name.
3. The additional characters need extra memory space, which you might need for longer programs.

The BASIC interpreter recognizes certain words as requests for specific operations. Names that are used to call up certain operations are called "reserved words." These words cannot be used as variable names because the interpreter will recognize the word as a request for the corresponding operation. Moreover, you cannot use a reserved word as any part of your variable name; BASIC will still find it and treat it as a request for an operation.

ARRAYS

An array is a sequence of related variables. A table of numbers, for example, may be visualized as an array. The individual numbers within the table become "elements" of the array.

Arrays are a useful shorthand means of describing a large number of related variables. Consider, for example, a table of numbers containing ten rows of numbers, with twenty numbers in

each row. There are 200 numbers in the table. How would you like it if you have to assign a unique name to each of the 200 numbers? It would be far simpler to give the entire table one name, and identify individual numbers within the table by their table location. That is precisely what an array does for you.

Arrays of up to eleven elements (subscripts 0 to 10 for a one-dimensional array) may be used where needed in VIC BASIC, just as variables can be used as needed. Arrays containing more than eleven elements need to be "declared" in a Dimension statement. Dimension statements are described later in this and in Chapter 3. An array (always with subscripts) and a single variable of the same name are treated as separate items by VIC BASIC. Once dimensioned, an array cannot be referenced with different dimensions.

OPERATORS

An operator is a special symbol that VIC BASIC recognizes as representing an operation to be performed on the variables or constant data. One or more operators, combined with one or more terms, form an "expression."

VIC BASIC provides arithmetic operators, relational operators, and Boolean operators.

An arithmetic operator defines an arithmetic operation to be performed on the adjoining terms. Arithmetic operations are performed using floating point numbers. Integers are converted to floating point numbers before an arithmetic operation is performed; the result is converted back to an integer. Consider the following operations and their symbols:

a. Addition (+). The plus sign specifies that the term on the left is to be added to the term on the right. For numeric quantities this is straightforward addition. Examples:

2 + 2
A + B + C
X% + 1
BR + 10E - 2

The plus sign can be used to add strings; but rather than adding their values, they are joined together, or concatenated, forming one longer string. The difference between numeric addition and string concatenation can be visualized as follows:

Addition numbers:
num1 + num2 = num3

Addition of strings:
string1 + string2 = string1string2

By concatenation, strings containing up to 255 characters can be developed.

EXAMPLES:

"FOR" + "WARD"	results in "FORWARD"
"HI" + "THERE"	results in "HITHERE"
A\$ + B\$	results in A\$B\$

b. Subtraction (-). The minus sign specifies that the term to the right of the minus sign is to be subtracted from the term to the left.

EXAMPLES:

$4 - 1$	results in 3
$100 - 64$	results in 36
$A - B$	results in the difference between the value of the two variables.
$55 - 142$	results in -87

The minus can also be used as a unary minus; that is, the minus sign preceding a negative number.

EXAMPLES:

-5
 $-9E4$
 $-B$
 $4 - -2$ same as $4 + 2$

c. Multiplication (*). An asterisk specifies that the term on the right is multiplied by the term on the left.

EXAMPLES:

$100 * 2$	results in 200
$50 * 0$	results in 0
$A * X1$	
$R \% * 14$	

d. Division (/). The slash specifies that the term on the left is to be divided by the term on the right.

EXAMPLES:

$10 / 2$	results in 5
$6400 / 4$	results in 1600
A / B	
$4E2 / XR$	

e. Exponentiation (\uparrow). The up arrow specifies that the term on the left is raised to the power specified by the term on the right. If the term on the right is 2, the number on the left is squared; if the term on the right is 3, the number on the left is cubed, etc. The exponent can be any number, variable, or expression, as long as the exponentiation yields a number in the VIC's range.

EXAMPLES:

$2 \uparrow 2$	results in 4
$12 \uparrow 2$	results in 144
$1 \uparrow 3$	results in 1

When an expression has multiple operations, as in:

$$A + C * 10 / 2 \uparrow 2$$

There is a built-in hierarchy for evaluating the expression. First, the exponentiation is considered, followed by the unary minus (−), followed by the multiplication and division (* /), followed then by the addition and subtraction (+ −). Operators of the same hierarchy are evaluated from left to right.

This natural order of operation can be overridden by the use of parentheses. Any operation within parentheses is performed first.

EXAMPLES:

$4 + 1 * 2$	results in 6
$(4 + 1) * 2$	results in 10
$100 * 4 / 2 - 1$	results in 199
$100 * (4 / 2 - 1)$	results in 100
$100 * (4 / (2 - 1))$	results in 400

When parentheses are present, VIC BASIC evaluates the innermost set first, then the next innermost, etc. Parentheses can be nested to any level and may be used freely to clarify the order of operations being performed in an expression.

A relational operator specifies a "true" or "false" relationship between adjacent terms. The specified comparison is made, and then the relational expression is replaced by a value of true (− 1) or false (0). Relational operators are evaluated after all arithmetic operations have been performed.

EXAMPLES:

$1 = 5 - 4$	results in true (− 1)
$14 > 66$	results in false (0)
$15 > = 15$	results in true (− 1)

Relational operators can be used to compare strings. For comparison purposes, the letters of the alphabet have the order $A < B$, $B < C$, $C < D$, etc. Strings are compared by comparing their stored character values. Characters are stored using a special

binary code called "ASCII." The Appendix lists the ASCII code assigned to every VIC character.

EXAMPLES:

"A"<"B" results in true (-1)
"X"="XX" results in false (0)
C\$=A\$+B\$

The Boolean operators AND, OR, and NOT specify a Boolean logic operation to be performed on two variables, on adjacent sides of the operator. In the case of NOT, only the term to the right is considered. Boolean operations are not performed until all arithmetic and relational operations have been completed.

EXAMPLES:

IF A= 100 AND B=100 THEN 10

If both A and B are equal to 100, branch to statement 10.

IF X<Y AND B>=44 THEN F=0

If X is less than Y, and B is greater than or equal to 44, then set F equal to 0.

IF A=100 or B=100 THEN 20

If either A or B has a value of 100, branch to statement 20.

IF X<Y OR B>=44 THEN F=0

F is set to 0 if X is less than Y, or B is greater than 43.

A single term being tested for "true" or "false" can be specified by the term itself, with an implied "<>0" following it. Any non-zero value is considered true; a zero value is considered false.

EXAMPLES:

IF A THEN B=2

IF A<>0 THEN B=2

The above two statements are equivalent.

IF NOT B THEN 100

Branch if B is false, i.e., equal to zero. This is probably better written as:

IF B=0 THEN 100

The three Boolean operators can also be used to perform logic operations on the individual binary digits of two adjacent terms (or just the term to the right in the case of NOT). But the terms must be in the integer range. Boolean operations are defined by groups of statements, which taken together constitute a "truth table." The following table lists the truth tables for the Boolean operators used by VIC BASIC.

Boolean Truth Table

The AND operation results in a 1 only if both bits are 1.

$$1 \text{ AND } 1 = 1$$

$$0 \text{ AND } 1 = 0$$

$$1 \text{ AND } 0 = 0$$

$$0 \text{ AND } 0 = 0$$

The OR operation results in a 1 if either bit is 1.

$$1 \text{ OR } 1 = 1$$

$$0 \text{ OR } 1 = 1$$

$$1 \text{ OR } 0 = 1$$

$$0 \text{ OR } 0 = 0$$

The NOT operation logically complements each bit.

$$\text{NOT } 1 = 0$$

$$\text{NOT } 0 = 1$$

This discussion of binary digit (bit) oriented Boolean operations is presented for those who are interested in the details of how these operations are performed. If you do not understand it, skip it. You are not skipping anything you must know. Recall that a single term has an implied "<>0" following it. The expression therefore becomes:

```
IF 0<>0 GOTO 40
```

Thus, the branch is not taken.

In contrast, a Boolean operation performed on two variables may yield any integer number:

```
IF A% AND B% GOTO 40
```

Assume that A% = 255 and B% = 240. The Boolean operation 255 AND 240 yields 240. The statement, therefore, is equivalent to:

IF 240 GOTO 40

or, with the "<>0":

IF 240 <>0 GOTO 40

Therefore the branch will be taken.

Now compare the two assignment statements*

A = A AND 10

A = A<10

In the first example, the current value of A is logically ANDed with 10 and the result becomes the new value of A. A must be in the integer range +32767 to -32768. In the second example, the relational expression A<10 is evaluated to -1 or 0, so A must end up with a value of -1 or 0.

LOGICAL OPERATORS

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision. For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen-bit, signed, two's-complement integers in the range -32768 to $+32767$. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1 , logical operators return 0 or -1 . The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For example, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will demonstrate how the logical operators work:

$63 \text{ AND } 16 = 16$ $63 = \text{binary } 111111 \text{ and } 16 = \text{binary } 10000$, so $63 \text{ AND } 16 = 16$

$15 \text{ AND } 14 = 14$ $15 = \text{binary } 1111 \text{ and } 14 = \text{binary } 1110$, so $15 \text{ AND } 14 = 14$ (binary 1110)

$-1 \text{ AND } 8 = 8$ $-1 = \text{binary } 11111111 \text{ and } 8 = \text{binary } 1000$, so $-1 \text{ AND } 8 = 8$

$4 \text{ OR } 2 = 6$ $4 = \text{binary } 100 \text{ and } 2 = \text{binary } 10$, so $4 \text{ OR } 2 = 6$ (binary 110)

$10 \text{ OR } 10 = 10$ $10 = \text{binary } 1010$, so $1010 \text{ OR } 1010 = 1010$ (10)

$-1 \text{ OR } -2 = -1$ $-1 = \text{binary } 11111111 \text{ and } -2 = \text{binary } 11111110$, so
 $-1 \text{ OR } -2 = -1$. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1 .

$\text{NOT } X = (-X) + 1$ The two's complement of any integer is the bit complement plus one.

STRING OPERATIONS

Strings may be concatenated using $+$. For example:

```
10 A$ = "FILE" : B$ = "NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

= <> < > <= >=

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

"AA" < "AB"

"FILENAME" = "FILENAME"

"X\$" > "X#"

"CL " > "CL"

"kg" < "KG"

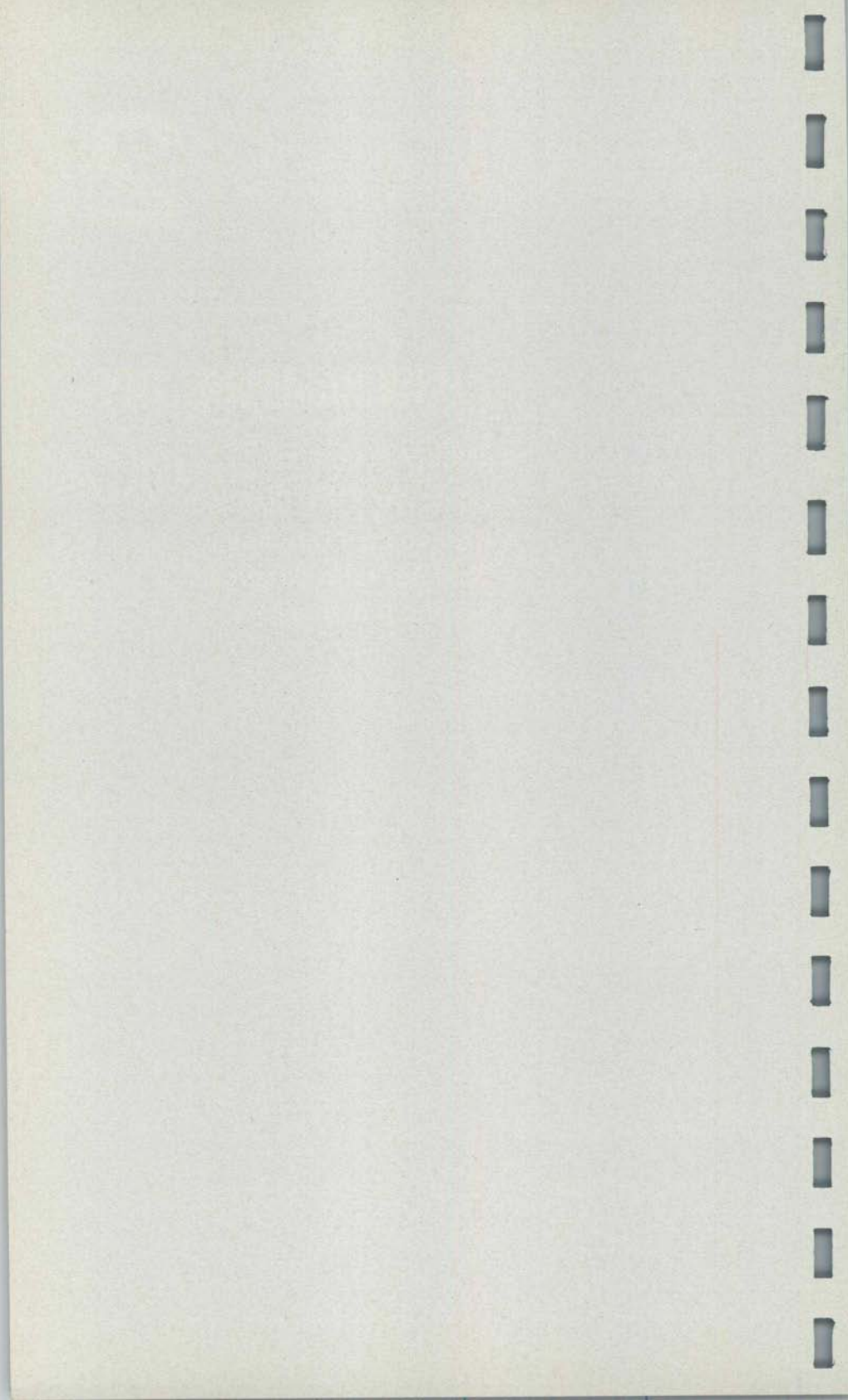
"SMYTH" < "SMYTHE"

B\$ < "9/12/78" where B\$ = "8/12/78"

2

PROGRAMMING TIPS

- Editing Programs
- Using the GET Statement
- Crunching Programs
- Working With Graphics
- Sound and Music



EDITING PROGRAMS

CURSOR CONTROLS

One of the most important features of the VIC is the ability to move the cursor around the screen and make changes to program lines. This is called "screen editing." The VIC will allow cursor movement around the screen in any direction, inserting extra spaces into a line, and erasing unwanted characters. The VIC's editor is one of the most powerful and easy to use of any computer.

There are 6 keys on the keyboard that are used for editing. These are the **CLR HOME**, **INST DEL**, **CRSR** (up), **CRSR** (down), **RETURN**, and the **SPACE** bar. These are all dual purpose keys, with different actions when the **SHIFT** or **C** keys are pressed. The **CRSR** (up), **CRSR** (down), **INST DEL**, and **SPACE** will all repeat if held down for at least a second.

a. **CLR HOME**

When unshifted, this moves the cursor to the upper left corner of the screen, called the "home position."


With the **SHIFT** (or **C** key) held down, this key will erase everything on the screen and move the cursor to the home position. This does not erase a program, variables, or anything else currently in memory.

b. **INST DEL**

Without the **SHIFT**, this key serves to delete the character to the left of the cursor. Anything else on the line to the right of the deleted character is moved one space to the left, filling in the gap.

With the **SHIFT**, this becomes an insert key. A space is created at the cursor's position, and everything else on the line to the right of the cursor is moved one more space to the right. If this pushes the last character on the line past the right end of the line, all lines below the current one are pushed down one line. Once the line is filled all the way to the end of 4 screen lines (88 characters), this key has no effect. (See also QUOTE MODE.)



This will move the cursor up or down one line on the screen, without affecting anything displayed on the screen. Unshifted, the cursor moves down, and if you hold down the **SHIFT** (or the  key) the cursor moves up.




This causes the cursor to move one space sideways. Unshifted, the move is to the right; and shifted, the cursor moves left. All characters remain on the screen without being erased. Notice that if the cursor moves beyond the *right* edge of the screen, it “wraps” one line down, to the left edge. If you move to the left edge the cursor wraps to the right side of the next line up (except from the home position).



The primary purpose of this key is to “enter” an instruction, calculation, or line of instructions. In direct mode the VIC executes the instruction or calculation. In program mode (when the instruction is preceded by a line number) the RETURN key causes the program line to be stored in memory. However, when you hold down the SHIFT key and hit RETURN, the VIC moves the cursor to the *next line* and the *left edge* of the screen but does not affect the line or instruction—SHIFT RETURN is a fast method for moving the cursor down the screen.



When you hit the **SPACE** bar (at the bottom of the keyboard) a blank space appears on the screen and the cursor moves one space to the right, erasing any character that was previously on that position.

If the **SHIFT** or  keys are held down while typing the **SPACE** bar, a character is printed that looks like a space but is actually treated by the VIC as a graphic character.

EDITING LINES

Anything displayed on the screen can be edited using the cursor controls. This can be a program line that is LISTed or typed in, or a command without a line number. To edit a line simply move the

cursor until it is on the line, then make the required changes, inserting or deleting as needed. Once you are finished, just hit the **RETURN** key. All the changes will be stored as made. If you just want to get the cursor past the line it is on, just hold **SHIFT** while hitting **RETURN** and the VIC will ignore the lines it passes.

In order to delete a line from a program, just type the line number and hit **RETURN**. There is no command for deleting more than 1 line at a time, although there is a trick for erasing all but a few lines. This involves LISTing the good portion of the program on the screen, typing NEW, moving the cursor to the top line displayed, and hitting **RETURN** on all the lines. This method only works if the section to be kept is very small (less than one screenful).

a. Direct Mode/"Calculator" Mode

If you enter a command or set of commands without a line number, they will execute as soon as you press the **RETURN** key. This is called "Direct Mode" or "Immediate Mode."

Since the VIC allows multiple statements on a line by using the colon (:), you can actually get a short program to run without entering line numbers. This is especially helpful when there is already a program in memory that you don't want to disturb. The maximum length of a program line is 4 screen lines, or 88 characters. In other words, if you enter a line numbered 10 you can display four 22-character lines on the screen but the VIC will store and interpret the information as one 88-character program line.

Here is a sample immediate mode program for a sound effect:

(don't hit **RETURN** until you've typed the whole line)

```
POKE 36878,15:FOR L=254 To 128 STEP-1:POKE 36876,  
L:POKE 36876,0:NEXT:POKE 36878,0
```

Some people also call this "calculator mode," because one of the most common things you can do with a VIC is perform calculations that don't require a program.

Here are some examples of calculations:

```
PRINT 5+4  
PRINT (2+4)*5/9  
PRINT SIN(37)
```

Certain commands can't be used in direct mode. These are: INPUT, INPUT#, GET, GET#, and DEF FN. The DATA statement

may be typed but has absolutely no effect. The first four commands shown can't be used because the same buffer that holds the statement being executed is also used to hold the characters being input. The DEF FN statement requires a line number so that the formula may be referenced later.

b. Program Mode

Instructions beyond a basic level of complexity require a *program*, as opposed to direct mode commands which can perform simple commands in a single line of instructions. *A program is one or more lines, each having its own unique number and containing a statement or group of statements.* Line numbers must be whole numbers from 0 to 63999. Programs are usually written using every 10th number, or even 100th number, since most programmers want to add more lines later between two existing lines, as the program is developed or edited.

Line numbers are stored in numerical order regardless of the order in which they are typed. The program will, when RUN, execute from the lowest to highest numbered lines, unless there is a command to jump to a different line, like GOTO, IF . . . THEN, or GOSUB.

USING THE GET STATEMENT

Most simple programs use the INPUT statement to get data from the person operating the computer. When dealing with more complex needs, such as protection from typing errors, the GET statement offers more flexibility and gives the program more intelligence. This section shows you how to use the GET statement to add some special screen editing features to your programs.

The VIC has a keyboard buffer that holds up to 10 characters. This means if the computer is busy doing some operation and is not reading the keyboard, you can still type in up to 10 characters, and the VIC will use them as soon as it finishes what it was doing.

This can be demonstrated with a simple program. Type in the program shown below. When you tell it to RUN, type the word HELLO on the keyboard. Since the VIC is busy in a loop, nothing appears on the screen—until the program stops after 15 seconds, and the word HELLO that you typed appears on the screen.

```
10 TI$ = "000000"  
20 IF TI$ < "000015" THEN 20
```

The VIC's input buffer is also called a queue, which is a good image to use to better understand how it works. Imagine standing in line waiting to buy a ticket to get into a movie. The first person in line is the first to get a ticket and leave the line, and the last person in line is the last to get a ticket. (In accounting, this is called the "first in, first out" method, or FIFO, as opposed to the "last in, first out", or LIFO method.)

The GET statement in the VIC acts as the ticket taker. First it looks to see if there are any characters "in line" (if a key or keys have been typed). If there are, the first character typed gets placed in a "variable" and out of the queue. If no characters are waiting in the buffer, then an empty value is returned.

One other point should be mentioned when talking about the queue. Any characters typed on the VIC's keyboard after the queue is full are lost, since the queue was full. So imagine that the ticket line is long enough to hold 10 people, and there is a cliff at the end of the line. Anyone trying to get into the line after the line is full simply falls off the cliff, never to be seen again.

Since the GET statement will keep going even when no character was typed, it is often necessary to put the GET statement into a *loop*, having it wait until the operator hits a key (actually, until a character has been received). Here is the recommended form:

(Type NEW to erase the previous program.)


```
10 GET A$ : IF A$ = "" THEN 10
```

There must be NO SPACE between the quotes in this line, to indicate an empty value. When the person is not typing anything, the empty value goes into the string variable (in this case represented by A\$ and the IF statement sends the program back to the GET statement. This loop repeats indefinitely, until the person operating the computer hits any key on the keyboard. At this point, the program continues with the line following line 10.

Add this line to the program:

```
100 PRINT A$; : GOTO 10
```

Now RUN the program. No cursor appears on the screen, but any character you type will be printed on the screen. This includes all special functions, like cursor and color controls and clearing the screen. This two-line program can be developed into a *screen editor*, shown below.

There are many features that you could use in a screen editor. A flashing cursor is nice to have, and can be programmed. Also, you may wish to "trap" certain keys, like  , so as not to erase the screen accidentally. Or you may wish to program the function keys for whole words. The following lines give each function key a special purpose. Remember, this is only the beginning of a program that you can customize for your needs, like word processing or data capture.

```
20 IF A$ = CHR$(133) THEN POKE 36879,8: GOTO 10
30 IF A$ = CHR$(134) THEN POKE 36879,27: GOTO 10
40 IF A$ = CHR$(135) THEN A$ = "DEAR SIR:" +
    CHR$(13)
50 IF A$ = CHR$(136) THEN A$ = "SINCERELY," +
    CHR$(13)
```

The CHR\$ numbers in the parentheses come from the CHR\$ code chart in the Appendix, which lists a different number for each key character. The four function keys are activated here to perform the tasks represented by the instructions which immediately follow the word THEN in each line . . . of course you could designate different keys by changing the CHR\$ number in parentheses, and different instructions after the THEN statement.

HOW TO CRUNCH BASIC PROGRAMS

You can pack more instructions—and power—into your BASIC programs by making each program as short as possible. This process of shortening programs is called “crunching.”

Crunching programs lets you squeeze the maximum possible number of instructions into your program. It also helps you reduce the size of programs which might not otherwise run in a given size; and if you're writing a program which requires the input of data such as inventory items, numbers or text, a short program will leave more memory space free to hold data.

But whether you're using an unexpanded VIC or a 32K VIC System, your programs will benefit from the following crunching techniques.

ABBREVIATING KEYWORDS

A list of keyword abbreviations is given in the Appendix A. This is helpful when you program because you can actually crowd more information on each line using abbreviations. The most frequently used abbreviation is the question mark (?) which is the BASIC abbreviation for the PRINT command. However, if you LIST a program that has abbreviations, the VIC will automatically print out the listing with the full-length keywords. If any program line exceeds 88 characters (4 lines on the screen) with the keywords unabbreviated, and you want to change it, you will have to re-enter that line with the abbreviations before saving the program. SAVEing a program incorporates the keywords without inflating any lines because BASIC keywords are tokenized by the VIC. Usually, abbreviations are added after a program is written and do not have to be LISTed any more before SAVEing.

SHORTENING PROGRAM LINE NUMBERS

Most programmers start their programs at line 100 and number each line at intervals of 10 (i.e., 100, 120, 130). This allows extra lines of instruction to be added (111, 112, etc.) as the program is developed. One means of crunching the program after it is completed is to *change the line numbers to the lowest numbers possible* (i.e., 1, 2, 3) because longer line numbers take more memory than shorter numbers. For instance, the number 100 uses

3 bytes of memory (one for each number) while the number 1 uses only 1 byte.

PUTTING MULTIPLE INSTRUCTIONS ON EACH LINE

You can put more than one instruction on each numbered line in your program by separating them by a colon. The only limitation is that all the instructions on each line, including colons, should not exceed the standard 88-character line length. Here is an example of two programs, before and after crunching:

BEFORE CRUNCHING:	AFTER CRUNCHING:
10 PRINT "HELLO. . .";	10 PRINT "HELLO. . .";:FOR
	T = 1 TO 500:NEXT:PRINT
20 FOR T = 1 TO 500:NEXT	"HELLO, AGAIN. . .":GOTO 10
30 PRINT "HELLO, AGAIN. . ."	
40 GOTO 10	

REMOVING REM STATEMENTS

REM statements are helpful in reminding yourself—or showing other programmers—what a particular section of a program is doing. However, when the program is completed and ready to use, you probably won't need those REM statements anymore and you can save quite a bit of space by removing the REM statements. If you plan to revise or study the program structure in the future, it's a good idea to keep a copy on file with the REM statements intact.

USING VARIABLES

If a number, word or sentence is used repeatedly in your program it's usually best to define those long words or numbers with a one or two letter variable. Numbers can be defined as single letters. Words and sentences can be defined as string variables using a letter and dollar sign. Here's one example:

BEFORE CRUNCHING	AFTER CRUNCHING
10 POKE 36878, 15	10 POKE 36878, 15: S = 36874
20 POKE 36874, 200	30 POKES, 200:POKES, 250:POKES,
30 POKE 36874, 250	150
40 POKE 36874, 150	

USING READ AND DATA STATEMENTS

Large amounts of data can be typed in as one piece of data at a time, over and over again . . . or you can print the instructional part of the program ONCE and print all the data to be handled in a long running list called the DATA statement. This is especially good for crowding large lists of numbers into a program.

USING ARRAYS AND MATRICES

Arrays and matrices are similar to DATA statements in that long amounts of data can be handled as a list, with the data handling portion of the program drawing from that list, in sequence. Arrays differ in that the list can be two or three dimensional.

ELIMINATING SPACES

One of the easiest ways to reduce the size of your program is to eliminate all the spaces. Although we often include spaces in sample programs to provide clarity, you actually don't need any spaces in your program and will save space if you eliminate them.

USING GOSUB ROUTINES

If you use a particular line or instruction over and over, it might be wise to GOSUB to the line from several places in your program, rather than write the whole line or instruction every time you use it.

USING TAB AND SPC

Instead of PRINTing several cursor commands to position a character on the screen, it is often more economical to use the TAB and SPC instructions to position words or characters on the screen.

WORKING WITH GRAPHICS

The graphics ability of the VIC 20 is more powerful and sophisticated than many users realize. The following material is a concept-by-concept guide to help you make better use of these graphics features to enhance your games and other programs.

CHARACTER MEMORY

Each character is formed in an 8-by-8 grid of dots, where each dot may be either "on" or "off." The character images are stored in a special chip called the "Character Generator ROM." The characters are stored as a set of 8 bytes for each character, with each byte representing the dot pattern of a row in the character, and each bit representing a dot. A zero (0) bit means that dot is off, and a one (1) bit means the dot is on.

The character memory in ROM begins at location 32768. The first 8 bytes contain the pattern for the @ sign, which has a character code value of zero on the screen. The next 8 bytes, from location 32776 to 32783, contain the information for forming the letter A.

IMAGE	BINARY	PEEK
* *	00011000	24
* *	00100100	36
* *	01000010	66
* * * * *	01111110	126
* *	01000010	66
* *	01000010	66
* *	01000010	66
* *	00000000	0

Each complete character set takes up 2K of memory, 8 bytes per character and 256 characters. Since there are two character sets, one for upper case and graphics and the other with upper and lower case, the character generator ROM takes up a total of 4K.

PROGRAMMABLE CHARACTERS

Since the characters are stored in ROM, it would seem like there is no way to change them for customizing characters. However, the memory location that tells the VIC where to find the characters is in a RAM location in the VIC chip, which can be changed to point to

many sections of memory. By changing the character memory pointer to point to RAM, the character set may be programmed for any need.

The VIC's standard characters are stored as follows:

HEX	DECIMAL	DESCRIPTION
8000	32768	Upper case with full graphics
8400	33792	Upper case & graphics—reversed
8800	34816	Upper and lower case with some graphics
8C00	35840	Upper & lower with some graphics—reversed

The register which controls where the chip gets its character information is at location 36869 decimal (9005 HEX). Its value is normally 240 (upper case and graphics) or 242 (upper/lower case).

The programmed character set cannot be put into expansion RAM, since the VIC chip doesn't have access to that memory. Therefore, any programmed characters must begin at a memory location between 4096 and 7168. Since BASIC programs are normally stored beginning at 4096, and strings start at the top of available memory and work their way down, precautions must be taken to protect the character set from being overwritten by BASIC. If the BASIC program begins at 4096, the normal procedure is to change the pointers to the top of available RAM at locations 52 and 56 so that they point below the character set. The following chart shows the possible locations of character sets, and the POKES to protect them.

Number	Location of Characters	Contents of Location	POKE 52 & 56
240	32768	Character ROM	
241	33792	Character ROM	
242	34816	Character ROM	
243	35840	Character ROM	
244	(36864)	VIC Chip, I/O	
245	(37888)	Color RAM	
246	(38912)	nothing	
247	(39936)	nothing	
248	(0)	Zero Page RAM	
249	(1024)	Expansion RAM	
250	(2048)	Expansion RAM	
251	(3192)	Expansion RAM	
252	4096	Start of BASIC RAM	

253	5120	BASIC RAM	20
254	6144	BASIC RAM	24
255	7168	BASIC RAM	28

This table assumes that screen memory starts at 7680 (1E00). However, it can be moved to other locations. The number of characters you have to work with at each location might change in that case.

There are two problems involved in creating your own special characters. First, it is an all or nothing process. Generally, if you use your own character set by telling the VIC chip to get the character information from the area you have prepared in RAM, the standard VIC characters are unavailable to you. To solve this problem, you must copy any letters, numbers, or standard VIC graphics you intend to use into your own character memory in RAM. You can pick and choose, take only the ones you want, and don't even have to keep them in order!

The second problem with programmable characters is that your character set takes memory space away from your BASIC program. This is a trade off situation, since you only have a limited amount of RAM available. If you decide to create a character set for a program, the program has to be smaller than a program which uses the standard VIC characters.

There are two locations in the VIC to start your character set that should not be used with BASIC—0 and 4096. The first should not be used because BASIC stores important data on page 0. The second can't be used because that is where your BASIC program starts! (If you expand your VIC, or use machine language, you can start your characters at 4096 if you want. This limit only applies to the unexpanded VIC.)

The best place to put your character set for use with BASIC while experimenting is at 7168. This is done by POKEing location 36869 with 255, giving you 64 characters to work with. Try the POKE now, like this:

POKE 36869,255

Immediately all the letters on the screen turn to garbage. This is because there are no characters set up at location 7168 right now . . . only random bytes. Set the VIC back to normal by using the RUN/STOP and RESTORE keys.

Now let's begin creating graphics characters. To protect your character set from BASIC, you should reduce the amount of memory BASIC thinks that it has. The amount of memory in your computer stays the same . . . it's just that you've told BASIC not to use some of it.

Type:

```
PRINT FRE(0)
```

The number displayed is the amount of memory space left unused. Now type the following:

```
POKE 52, 28: POKE56, 28: CLR
```

Now type:

```
PRINT FRE(0)
```

See the change? BASIC now thinks it has 512 bytes less memory to work with. The memory you just reclaimed from BASIC is where you are going to put your character set, safe from actions of BASIC.

The next step is to put your characters into RAM. When you begin, there is random data beginning at 7168. You must put character patterns in RAM (in the same style as the ones in ROM) for the VIC to use.

The following one line program moves 64 characters from ROM to your character set RAM:

```
FOR I= 7168 TO 7679: POKE I, PEEK(I+25600): NEXT
```

Now POKE 36869 with 255. Nothing happens, right? Well, almost nothing. The VIC is now getting its character information from your RAM, instead of from ROM. But since we copied the characters from ROM exactly, no difference can be seen . . . yet.

You can easily change the characters now. Clear the screen and type an @ sign. Move the cursor down a couple of lines, then type:

```
FOR I = 7168 TO 7168+7:POKE I, 255 - PEEK(I) : NEXT
```

You just created a reversed @ sign!



VIC TIP: Reversed characters are just characters with their bit patterns in character memory reversed!

Now move the cursor up to the program again and hit return again to re-reverse the character (bring it back to normal). By looking at the table of screen display codes, you can figure out where in RAM each character is. Just remember that each character takes eight memory locations to store.

Here are a few examples just to get you started:

CHARACTER	DISPLAY CODE	CURRENT STARTING LOCATION IN RAM
@	0	7168
A	1	7176
!	33	7432
>	62	7664

Remember that we only took the first 64 characters, though. Something else will have to be done if you want one of the other characters.

What if you wanted character number 154, a reversed Z? Well, you could make it yourself, by reversing a Z, or you could copy the set of reversed characters from the ROM, or just take the one character you want from ROM and replace one of the characters you have in RAM that you don't need.

Suppose you decide that you won't need the > sign. Let's replace the > sign with the reversed Z. Type this:

```
FORI=7664 TO 7671: POKE I, PEEK(I+26336): NEXT
```

Now type a > sign. It comes up as a reversed Z. No matter how many times you type the >, it comes out as a reversed Z. (This change is really an illusion. Though the > sign looks like a reversed Z, it still acts like a > in a program. Try something that needs a > sign. It will still work fine, only it will look strange.)

A quick review: We can now copy characters from ROM into RAM. We can even pick and choose only the ones we want. There's only one step left in programmable characters (the best step!) . . . making your own characters.

Remember how characters are stored in ROM? Each character is stored as a group of eight bytes. The bit patterns of the bytes directly control the character. If you arrange 8 bytes, one on top of another, and write out each byte as eight binary digits, it forms an eight-by-eight matrix, looking like the characters. When a bit is a one, there is a dot at that location. When a bit is a zero, there is a space at that location.

When creating your own characters, you set up the same kind of table in memory. Type this program:

```
10 FORC= 7328 TO 7335: READ A: POKE C,A: NEXT
20 DATA 60, 66, 165, 129, 165, 153, 66, 60
```

Now type RUN. The program will replace the letter T with a smile face character. Type a few T's to see the face. Each of the numbers in the DATA statement in line 20 is a row in the smile face character. The matrix for the face looks like this:

	7	6	5	4	3	2	1	0	DECIMAL	BINARY
ROW 0			*	*	*	*			60	00111100
1		*					*		66	01000010
2	*		*			*		*	165	10100101
3	*							*	129	10000001
4	*		*			*		*	165	10100101
5	*			*	*			*	153	10011001
6		*					*		66	01000010
ROW 7			*	*	*	*			60	00111100

The sheet on this page will help you design your own characters. There is an 8-by-8 matrix on the sheet, with row numbers, and numbers at the top of each column. (If you view each row as a binary word, the numbers are the value of that bit position. Each is a power of 2. The leftmost bit is equal to 128 or 2 to the 7th power, the next is equal to 64 or 2 to the 6th, and so on, until you reach the rightmost bit (bit 0) which is equal to 1 or 2 to the 0th power.)

Place an X on the matrix at every location where you want a dot to be in your character. When your character is ready you can create the DATA statement for your character.

Begin with the first row. Wherever you placed an X, take the number at the top of the column, and write it down. When you have the numbers for every column of the first row, add them together. Write this number down, next to the row. This is the number that you will put into the DATA statement to draw this row.

Do the same thing with all of the other rows (1-7). When you are finished you should have 8 numbers between 0 and 255. If any of your numbers are not within range, recheck your addition. The numbers must be in this range to be correct! If you have less than 8 numbers, you missed a row. It's OK if some are 0. The 0 rows are just as important as the other numbers.

	7	6	5	4	3	2	1	0
0								
1								
2								
3								
4								
5								
6								
7								

Replace the numbers in the DATA statement in line 20 with the numbers you just calculated, and RUN the program. Then type a T. Every time you type it, you see your own character!

If you don't like the way the character turned out, just change the numbers in the DATA statement and re-RUN the program until you are happy with your character.

That's all there is to it!

HIGH RESOLUTION GRAPHICS

When writing games or other types of programs, sooner or later you get to the point at which you want a high resolution display, or smooth movement of objects on the screen. A regular character can move one space at a time, which is 8 rows or columns of dots. For smoother movement, characters should be moved one row of dots at a time, using high-resolution graphics.

The VIC can handle this need: high resolution is available through bit mapping the screen. Bit mapping is the name of the method where each possible dot (pixel) of resolution on the screen is assigned its own bit in memory. If that memory bit is a one, the dot it is assigned to is on. If the bit is set to zero, the dot is off. You can bit map the entire screen of the VIC, or only a portion of it. You can mix HI-RES, programmable characters and regular graphics.

High resolution has a few drawbacks, which is why it is not used all the time. It takes a lot of memory to bit map the entire screen. Because every pixel must have a memory bit to control it, you are going to need one bit of memory per pixel (or one byte for 8 pixels). Since each character is 8-by-8, and there are 23 lines of 22 characters, the resolution is 176 by 184 for the whole screen. That gives you 32384 separate dots, each of which requires a bit in memory, or 4048 bytes of memory needed to map the whole screen.

Fear not, you can still use high resolution graphics on the unexpanded VIC! You just don't bit map the entire screen. Instead, you bit map just as much of the screen as you have memory for, and either use the rest of the screen as a border, or use it for text. A 64 dot by 64 dot screen section will be fairly easy to work with for this section.

Generally, high resolution operations are made of many short, simple, repetitive routines. Unfortunately, this kind of thing is rather slow using BASIC, so high resolution routines written in BASIC are usually rather slow. However, short, simple, repetitive routines are exactly what machine language does best. The solution is to either write your programs entirely in machine language (painful), or call

HI-RES subroutines from your BASIC program, using the SYS command from BASIC. That way you get both the ease of writing in BASIC, and the speed (for graphics) of machine language. The SUPER-EXPANDER cartridge also is available to add HI-RES commands to VIC BASIC.

All of the examples given in this section will be in BASIC to make them clear. In the future, you can add the routines to your own programs to give you easy HI-RES graphics. Now to the technical details.

Remember programmable characters? Well, bit mapping is done almost the same way. When you created a programmable character, you could watch it form before your eyes, if the character was on the screen when you were changing it.

To see this again, type this program in and RUN it.

```
100 POKE 36869 , 255
110 FOR I = 7168 TO 7679 : POKE I, PEEK(I + 25600) : NEXT
120 PRINT CHR$(147) "A"
130 FOR I = 7176 TO 7183
140 FOR L = 1 TO 1000 : NEXT
150 READ A : POKE I, A : NEXT
160 DATA 60,36,36,36,36,255,255
```

The character changed from an A to a top hat as you watched! This is the trick behind HI-RES graphics on the VIC—making changes directly on the character memory. When the character is already on the screen you see the changes right away!

The best way to set up the HI-RES display screen for the 64-dot by 64-dot HI-RES display is to print out 64 characters in a square box or matrix.

Setting up the HI-RES display screen is the first step in HI-RES graphics. The following short program section will set up the display screen.

Type NEW, then

```
10 POKE 36879,8:PRINT CHR$(147)
20 FOR L = 0 TO 7 : FOR M = 0 TO 7
30 POKE 7680 + M*22 + L, L*8 + M
40 NEXT : NEXT
```

Now RUN the program. We now have 64 characters on the screen; they can't be changed in any way, since the codes for the letters are in ROM. If we change the character memory register to

point to RAM, however, we will be displaying memory that we can change any way we want.

Add the following line to the program:

```
5 POKE 36869,255
```

(This will give us 64 programmable characters, which we can set up as an 8-by-8 character matrix, which will give us a 64-dot by 64-dot HI-RES screen—just what we were looking for.)

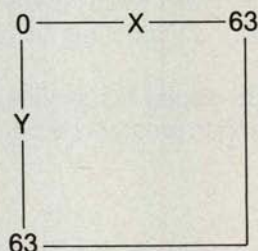
RUN the program now. Garbage appeared on the screen, right? Just like the regular screen, we have to clear the HI-RES screen before we use it. Printing a key won't work in this case. Instead we have to clear out the section of memory used for our programmable characters. Add the following line to your program to clear the HI-RES screen:

```
6 FOR I= 7168 TO 7679 : POKE I, 0 : NEXT
```

Now RUN the program again. You should see nothing but black on the screen—your program is a success. What we want to add now is the means to turn dots on and off on the HI-RES screen.

To SET a dot (turn the dot on) or UNSET a dot you must know how to find the correct bit in the character memory that you must set to one. In other words, you have to find the character you need to change, the row of the character, and which bit of the row that you have to change. We need a formula to calculate this.

We will use X and Y to stand for the horizontal and vertical position of the dot. The dot where X=0 and Y=0 is at the upper-left of the display. Dots to the right have higher X values, and the dots toward the bottom have higher Y values.



The dots where $0 \leq X \leq 7$ and $0 \leq Y \leq 7$ are in character number 0, which we placed at the upper-left corner of the screen. Each character contains 64 dots, 8 rows of 8 dots each.

These are the simple calculations to decide which dot of which character is being altered:

The character number is. . .

$\text{CHAR} = \text{INT}(X/8) * 8 + \text{INT}(Y/8)$

This gives the display code of the character you want to change. To find the proper row in the character, use this formula:

$$\text{ROW} = (\text{Y}/8 - \text{INT}(\text{Y}/8)) * 8$$

Therefore, the byte in which character memory dot (X,Y) is located is calculated by:

$$\text{BYTE} = 7168 + \text{CHAR} * 8 + \text{ROW}$$

The last thing we have to calculate is which bit should be modified. This is given by:

$$\text{BIT} = 7 - (\text{X} - \text{INT}(\text{X}/8) * 8)$$

To turn on any bit on the grid with coordinates (X,Y), use this line:

$$\text{POKE BYTE, PEEK (BYTE) OR } (2 \uparrow \text{BIT})$$

Let's add these calculations to the program. In the following example, the VIC will plot a sine curve:

```
50 FOR X = 0 TO 63
60 Y = INT( 32 + 31 * SIN (X/10))
70 CH = INT(X/8)*8 + INT(Y/8)
80 RO = (Y/8 - INT(Y/8)) * 8
90 BY = 7168 + 8*CH + RO
100 BI = 7 - (X - INT(X/8)*8)
110 POKE BY, PEEK(BY) OR (2 ↑ BI)
120 NEXT
130 GOTO 130
```

The calculation in line 60 will change the values for the sine function from a range of +1 to -1 to a range from 0 to 63. Lines 70 to 100 calculate the character, row, byte, and bit being affected, using the formulas as given before. Line 130 freezes the program by putting it into an infinite loop. When you have finished looking at the display, just hold down RUN/STOP and hit RESTORE.

As a further example, you can modify the sine curve program to display a circle. Here are the lines to type to make the changes:

```
55 Y1 = 32 + SQR(64*X - X*X)
56 Y2 = 32 - SQR(64*X - X*X)
60 FOR Y = Y1 TO Y2 STEP Y2 - Y1
125 NEXT
```

This will create a circle in the HI-RES area of the screen. Notice that the rest of the screen fills up with a stripe pattern. This is because the empty spaces on the screen are filled with character code 32, which is normally a space—and is now one of the

programmable characters in the grid. If you didn't want the screen to fill up with that garbage, just fill the screen with characters with a code you're not using. In this case, code 160 would work nicely, since that points to the blank space character in ROM. Here is a line that cleans up the rest of the screen:

```
11 FOR I = 7680 TO 8185 : POKE I,160 : NEXT
```

MULTI-COLOR MODE GRAPHICS

High resolution graphics gives you control of very small dots on the screen. Each dot in character memory can have 2 possible values, 1 for on and 0 for off. When a dot is off, the dot on the screen is drawn with the screen color. If the dot is on, the dot is colored with the character color for that screen position. All the dots within each 8×8 character can either have the screen color or the character color. This limits the color resolution within that space.

Multi-color mode gives a solution to this problem. Each dot in multi-color mode can be one of 4 colors: screen color, character color, border color, or auxiliary color. The only sacrifice is in the horizontal resolution, because each multi-color mode dot is twice as wide as a high-resolution dot. This loss of resolution is more than compensated for by the extra abilities of multi-color mode, like the ability to color dots in one of 16 colors, instead of the usual 8.

Multi-color mode is set on or off for each space on the screen, so that multi-color graphics can be mixed with high-resolution graphics. This is controlled in color memory. Color memory is in locations beginning at either 37888 or 38400, depending on the size of memory in the VIC. To find the current location of color memory, use the formula:

$$C = 37888 + 4 * (\text{PEEK}(36866) \text{ AND } 128)$$

The memory in this location is a little different from that in the rest of the VIC. It is wired up as nibbles instead of bytes, meaning that each memory location has 4 bits instead of the usual 8. When PEEKing values from this section of memory, the value should always be ANDed with 15 to "filter out" any random bits that appear in the upper 4 bits.

By POKEing a number into color memory, you can change the color of the character in that position on the screen. POKEing a number from 0 to 7 gives the normal character colors. POKEing a number between 8 and 15 puts the space into multi-color mode. In other words, turning the high bit on in color memory sets multi-color mode, and turning off the high bit in color memory sets normal (or high-resolution) mode.

Once multi-color mode is set in a space, the bits in the character

determine which colors are displayed for the dots. For example, here is a picture of the letter A, and its bit pattern:

IMAGE	BIT PATTERN
-------	-------------

* *	00011000
* *	00100100
	* 01000010
* * * * *	01111110
* *	* 01000010
* *	* 01000010
* *	* 01000010
	00000000

In normal or high-resolution mode, the screen color is displayed everywhere there is a 0 bit, and the character color is displayed where the bit is a 1. Multi-color mode uses the bits in pairs, like so:

IMAGE	BIT PATTERN
-------	-------------

AABB	00 01 10 00
BBAA	00 10 01 00
AA BB	01 00 00 10
AACCCBB	01 11 11 10
AA BB	01 00 00 10
AA BB	01 00 00 10
AA BB	01 00 00 10
	00 00 00 00

In the image area above, the spaces marked AA are drawn in the border color, the spaces marked BB use the character color, and the spaces marked CC use the auxiliary color. The bit pairs determine this, according to the chart below:

BIT PAIR	COLOR REGISTER
00	Screen color
01	Border color
10	Character color
11	Auxiliary color

Turn the VIC off and on, and type this demonstration program:

```

100 C = 37888 + 4 * (PEEK (36866) AND 128)
110 POKE 36878, 11 * 16 : REM SET AUX COLOR
120 PRINT CHR$(147) "AAAAAAAAAA"
130 FOR L = 0 TO 9

```

140 POKE C + L , 8

150 NEXT

The screen color is white, the border color is cyan, the character color is black, and the auxiliary color is light cyan.

You're not really putting color codes in the space for screen, border, and auxiliary color; you're putting references to the registers associated with those colors. This conserves memory, since 2 bits can be used to pick 16 colors (screen and auxiliary) or 8 colors (character and border). This also makes some neat tricks possible. Simply changing one of the indirect registers will change every dot drawn in that color. So everything drawn in the screen, border, or auxiliary color can be changed on the whole screen instantly. Here is an example using the auxiliary color:

```
100 PRINT CHR$(147) CHR$(18);
110 POKE 646 , 8
115 FOR L = 1 TO 22 : PRINT CHR$(32); : NEXT
120 POKE 646 , 6
130 PRINT "HIT A KEY"
140 GET A$: IF A$ = "" THEN 140
150 X = INT (RND (1) *16)
160 POKE 36878 , X * 16
170 GOTO 140
```

There is a memory location in the VIC that is especially useful with multi-color mode. Location 646 is the color that is currently being PRINTed. When a color control key is pressed this location is changed to the new color code. By POKEing this location, the characters to PRINT can be changed to any color, including multi-color characters. For example, type this command:

```
POKE 646,10
```

The word READY and anything else you type will be displayed in multi-color mode. Any color control will set you back to regular text.

SUPEREXPANDER CARTRIDGE

There is a cartridge program called the VIC SUPER EXPANDER. This cartridge is programmed with many special functions, including a graphics package. This allows drawing of lines, dots, and circles, coloring in of areas on the screen, and full control over graphic modes. For programs in BASIC, it will be considerably easier to use graphics with the SUPER EXPANDER than by use of cumbersome pokes. The SUPER EXPANDER also includes 3K of extra RAM to give you enough room to do any high-resolution operation.

SOUND AND MUSIC

Sound effects and music can improve almost any computer program, whether in BASIC or Machine Language. Obviously, a computer game is more exciting if you can hear the guns blazing and the rockets exploding. Likewise, a clever little tune provides an audio "theme" for a game or other program, or might become the "reward" if the player reaches a special "high" score.

Beyond games, sound effects serve other useful purposes. For example, a business or calculation program may be faster and easier to use if the computerist can enter a long string of numbers or formulas without looking up from a chart or balance sheet. A quick tone at the end of each entry indicates when a number has been entered . . . a "buzz" might sound if the number entered has too many decimal places . . . and different tones might be used to distinguish one kind of entry from another.

These are just a few ideas about how sound and music are used in computer programming. The following information is provided to help you the programmer understand how to use the VIC's sound capability to best advantage.

FOUR "SPEAKERS" AND 5 "OCTAVES"

The VIC has 3 tone generators (for music), and one white noise generator (for sound effects). The tone generators cover 3 octaves each but are staggered slightly so you can actually reach a total of 5 separate octaves.

The VIC's speakers and volume control are stored in specific memory locations which you can access and control by using the POKE command. Whenever you poke one of these locations you activate that tone generator, or the volume control.

When programming sound—especially music—it is often helpful to think of these various sound controls as "speakers," and the volume setting as a standard "volume" control.

Here, briefly, is a list of memory locations relating to sound:

- 36878 (VOLUME SETTING)
- 36874 (SPEAKER 1—MUSIC—LOWEST)
- 36875 (SPEAKER 2—MUSIC—MIDDLE)
- 36876 (SPEAKER 3—MUSIC—HIGHEST)
- 36877 (SPEAKER 4—NOISE)

There are 15 volume settings. Therefore to set the volume you must type the POKE command, followed by a comma, and a number

from 1 to 15. We suggest you always use 15, unless you're playing with the amplitude as part of a sound effect.

Each speaker has a range of 128 separate settings. To "play" a particular note you must POKE one of the speaker settings, which happen to be numbered from 128 to 255. If you POKE a number lower than 128 or higher than 255 you will get no sound (which suggests one way to interrupt a speaker while it's "on").

Here's an example of how to play a note on the VIC:

```
20 POKE 36878,15      SETS VOLUME AT MAXIMUM
30 POKE 36875,200     TURNS ON SPEAKER NUMBER 2
40 FOR X=1TO1000:     THIS IS A 1000 COUNT TIME DELAY
NEXT
50 POKE 36878,0       THIS TURNS THE SPEAKER OFF
                      AFTER COUNTING TO 1000
```

RUN

The VIC uses the television speaker as its "voice," so the volume can also be adjusted by turning the television speaker (or other external amplifier).

ABBREVIATING THE SOUND COMMANDS

You can abbreviate the lengthy POKE commands described above by converting these to programming "shorthand." One way is shown below:

```
10 V=36878:S1=36874:S2=36875:S3=36876:S4=36877
```

Now if you want to turn on a particular speaker, or set volume, you can use the abbreviations . . . like, for instance:

```
20 POKEV,15
30 POKES2,200
40 FORX=1TO1000:NEXT
50 POKEV,0
```



VICTIP

In line 10, we put all the commands on one line instead of five lines because we want to demonstrate economical programming techniques. You can save memory and "crunch" longer programs into less space if you put several commands on a single line, with each command separated by a colon, as shown.

Keep this "programming shorthand" in mind as we show you some more examples of using the VIC speakers.

HOW MUSIC WORKS ON THE VIC

As already mentioned, the VIC's speakers each cover 3 octaves, but together reach a total range of 5 octaves. This is because the VIC's 3 tone generators are "staggered" so the octaves of the different speakers overlap. A more graphic picture of which speakers cover which octaves and how they overlap is shown in the chart on page 99. Musical note values are shown below:

TABLE OF MUSICAL NOTES

APPROX. NOTE	VALUE	APPROX. NOTE	VALUE
C	135	G	215
C#	143	G#	217
D	147	A	219
D#	151	A#	221
E	159	B	223
F	163	C	225
F#	167	C#	227
G	175	D	228
G#	179	D#	229
A	183	E	231
A#	187	F	232
B	191	F#	233
C	195	G	235
C#	199	G#	236
D	201	A	237
D#	203	A#	238
E	207	B	239
F	209	C	240
F#	212	C#	241

SPEAKER COMMANDS:	WHERE X CAN BE:	FUNCTION:
POKE 36878,X	0 to 15	sets volume
POKE 36874,X	128 to 255	plays tone
POKE 36875,X	128 to 255	plays tone
POKE 36876,X	128 to 255	plays tone
POKE 36877,X	128 to 255	plays "noise"

The Octave Chart illustrates the three octaves contained in each speaker register. It also shows how several octaves overlap . . . for instance, the lowest octave of Speaker 3 contains the same notes as the middle octave of Speaker 2. Of course, the same note played on different speakers may sound slightly different . . . just as the same note played on a piano may sound different from the same note played on a harpsichord. Also, some television sets and speakers may cause varying results in terms of tonal qualities.

The Table of Musical Notes on page 97 is intended to help you approximate note values in your computer program using the VIC. The number values are approximate only and may be adjusted by using values between those shown.

MUSIC PROGRAMMING TECHNIQUES

There are four basic parameters in programming music:

1. Volume
2. Speaker/Sound Register Selection
3. Note
4. Duration

In other words, the things you have to consider when programming music are which volume to set, which speaker(s) to use, the notes being played by each speaker, and the duration of each note. Let's consider some techniques for putting these parameters in your program:

EXAMPLE 1: MUSIC USING DATA STATEMENT

10 POKE 36878, 15	Set volume to highest level (15).
20 S2 = 36875	Set speaker to equal S2 (any variable).
30 READ N,D	Read duration & note from DATA below.
40 IF N = -1 THEN POKE S2,0:END	Turn off speaker & end program at -1.
50 POKE S2,N	Play note N from DATA on Speaker S2.
60 FOR T = 1 TO D: NEXT	Duration loop to set up time value.
70 GOTO 30	Keeps going back to DATA list to get duration & note (N,D) values.
80 DATA 225,250,226, 250,227,250,228, 250,229,250,230, 250,231,250,232, 250,233,250,234, 250,235,250, -1, -1.	DATA statements . . . the first number is the note from the note value chart earlier in text, and the second number is the duration the note is played.

VIC OCTAVE COMPARISON CHART

S3 (36876)

D O
E C
F T
G A
A V
B E
C 3

HIGHEST OCTAVE

S2 (36875)

D O
E C
F T
G A
A V
B E
C 3

S1 (36874)

D O
E C
F T
G A
A V
B E
C 3

D O
E C
F T
G A
A V
B E
C 1

D O
E C
F T
G A
A V
B E
C 1

D O
E C
F T
G A
A V
B E
C 2

D O
E C
F T
G A
A V
B E
C 1

LOWEST OCTAVE

Pay special attention to the fact that N,D is actually each PAIR of values in the DATA statement in line 80. For example, 225,250 is the note (225) and time duration (250 jiffies) the note is played. The next note is 226 which is also held for a duration of 250, and so on until the computer comes to the DATA pair of - 1, - 1, which is the signal to END the program. There are two ways to end the music portion of a program. To *turn off the music and continue the program* you should simply POKE the speaker(s) with a zero to turn them off when the program reaches your DATA signal (the signal here is - 1, - 1 but it could be any number). To *end the entire program when the music stops* POKE the speaker(s) off with a zero and END the program by using the END command as shown in line 40.

EXAMPLE 2: MUSIC USING MULTIPLE SPEAKERS

```

10 POKE 36878,15
20 S1=36874:S2=36875:
   S3=36876
30 READ D,N1,N2,N3
40 IF D=-1THENPOKES1,
   0:POKES2,0:POKES3,
   0:END
50 POKE S1,N1
60 POKE S2,N2
70 POKE S3,N3
80 FORT=1TOD:NEXT
90 GOTO30
100DATA500,225,225,225,
   0,0,0,0,500,225,225,225,
   500,232,232,232,0,0,0,0,
   500,232,232,232
110DATA250,240,240,240,
   250,239,239,239,
   125,237,237,237,
   67.5,235,235,235,
   33,232,232,232
120DATA33,231,231,231,
   33,228,228,228,
   33,225,225,225,
   33,223,223,223,
   500,195,195,195
130DATA500,240,240,240
140DATA-1,0,0,0

```

This program is essentially the same as Example 1, except several speakers are used, and each speaker must be designated separately. If you're not familiar with DATA statements, here's a good example of how they work. In the *previous* example we told the VIC to scan through the DATA and READ N,D where the first number was the NOTE and the second number was the DURATION. In this example, we are rearranging the program so the VIC reads the DATA numbers in a slightly different order. Line 30 instructs the VIC to READ the data in this order: DURATION, NOTE 1, NOTE 2, NOTE 3. These notes are only played on their corresponding speakers per lines 60-70. The music is the same as EXAMPLE 1 except we're using three speakers and setting the same duration for all 3 notes so they play simultaneously. If you want different speakers to play

different durations, (as in most songs) you would change line 30 to: READ D1,N1,D2,N2,D3,-N3 and put matching duration/note pairs in the DATA statements. This is how you achieve 3-voice harmony.

Notice also in Example 2 that the simple tune is played with all similar notes, then speeded up by shortening the *duration values*. The duration values may be any number, including decimal numbers like the "67.5" in line 110 which is included as an example. The note values may be any number between 128 and 255, with notes corresponding to the note value chart earlier in this chapter.

Another parameter which you might consider changing is volume. An example of how volume may be changed is found in the "OCEAN WAVES" sound effect program on page 137 of the VIC 20 Personal Computer Guide (owner's manual).

If you want to get even more sophisticated in your music/sound effects programming, try *frequency modulation*, which entails rapidly switching back and forth between two notes to achieve the illusion of a "middle" note between the two values. Example 3 illustrates this technique and plays a "true" scale.

EXAMPLE 3: TRUE NOTE SCALE USING FREQUENCY MODULATION

READY.
MUSE

READY.

```

90 S1 = 36874:S2 = S1 + 1:S3 = S1 + 2:V = S1 + 4
100 DIMN(37,1):FORI = 0TO37:READN(I,0),N(I,1):NEXT
200 FORI = 0TO37:POKEV,15:FORJ = 0TO49:POKES1,N(I,0):
POKES1,N(I,1):NEXTJ:POKEV,0:NEXTI
9000 DATA131,131,140,140,145,145,151,151,158,158,161,162,
166,167,173,174,178,178,181,182
9010 DATA185,186,189,190,192,195,197,197,200,200,203,203,
206,207,208,209,211,212,214,214
9020 DATA216,216,218,219,220,221,222,223,224,224,226,226,
227,228,229,229,231,231,232,232
9030 DATA233,233,234,235,235,236,237,237,237,238,239,239,
239,240,240,241
READY.
```

MUSICAL NOTE VALUES

The accompanying chart shows the two values to modulate between to get the "true" note in the first column. Using the program in Example 3 above, POKE the first value, then the second value in line 100 to get the "true" modulated tone.

NOTE	VALUE 1	VALUE 2
C	131	
C#	140	
D	145	
D#	151	
E	158	
F	161	162
F#	166	167
G	173	174
G#	178	
A	181	182
A#	185	186
B	189	190
C	192	195
C#	197	
D	200	
D#	203	
E	206	207
F	208	209
F#	211	212
G	214	
G#	216	
A	218	219
A#	220	221
B	222	223
C	224	
C#	226	
D	227	228
D#	229	
E	231	
F	232	
F#	233	
G	234	235
G#	238	236
A	237	
A#	237	238
B	239	

C	239	240
C#	240	241

If two note values are given, vary the sound register on the VIC between those two values. If only one value is given, don't vary the register (just POKE the value in twice).



VIC TIP: Here are a few additional comments about using DATA statements in your programs. In Example 2 (page 100) we show each note set (duration, note 1, 2 and 3) on a separate line to emphasize that the notes are arranged and played simultaneously ... when you enter this DATA in your computer, however, you should not break up the segments but instead type all the numbers and commas without any spaces between the characters. Typing programs without spaces is a good way to conserve memory and reduces the possibility of error.

EXAMPLE 4: THE VIC PIANO

Finally, to give you a more familiar representation of how music works on the VIC, here's a program which converts the VIC keyboard to a "piano."

```

10 REM STORE SOUND REGISTERS
20 S2=36875
30 V =36878
40 POKE S2,0

100 REM STORE B MAJOR SCALE
110 FOR N=1 TO 8
120 READ A (N)
130 NEXT N

140 DATA 223, 227, 230
150 DATA 231, 234, 236
160 DATA 238, 239

200 REM PLAY KEYBOARD
210 POKE V, 15

220 GET A$: IF A$="" THEN 220
230 N=VAL (A$)

```

Abbreviates the voice registers we'll need and turns them off

Reads B major scale from lines 140-160

Contains POKE values for B major scale

Turns on the volume

Finds out what key is being pressed

240 IF N=0 OR N=9 THEN 300

250 POKE S2,0

260 FOR T=1 TO 25: NEXT T

270 POKE S2, A (N)

280 GOTO 220

300 REM ENDING MODULE

310 POKE S2, 0

Ends the program
if you've pressed
"0" or "9"

Brief silent interval
between notes. . .

Plays the tone
and returns to
look for another

Turn off the sound
before you go

Now, when you type RUN (and press **RETURN**), you can play tunes on your VIC. The keys in the top row with numbers on them control the various notes:

1	2	3	4	5	6	7	8
DO	RE	MI	FA	SOL	LA	TI	DO

The VIC will keep playing the note you hit last until you hit another note. When you're done, press either 0 or 9, and it will turn off. To start the VIC piano again, just reRUN the program.

Try the following:

1 1 5 5 6 6 5

4 4 3 3 2 2 1

5 5 4 4 3 3 2

5 5 4 4 3 3 2

1 1 5 5 6 6 5

4 4 3 3 2 2 1 8

9

OR:

3 3 4 5 5 4 3 2

1 1 2 3 3 2 2

3 3 4 5 5 4 3 2

1 1 2 3 2 1 1

0

THE WHITE NOISE GENERATOR

One of the "speakers" we've ignored thus far is the White Noise Generator, or Speaker 4. This fourth speaker produces a blank noise sound like that on your television set when you fall asleep late at night. It has the same 3 octave range as the tone generators described above, and is used primarily for sound effects, either by itself or in conjunction with the other speakers. The combination of white noise and tones can produce some stunning effects. Twenty or so sample sound effects are listed in the VIC 20 owner's manual (PERSONAL COMPUTING ON THE VIC 20).

To try out the White Noise generator, try typing this:

10 POKE36878,15 (if you don't have vol on yet)

20 POKE36877,240

30 FORT=1TO1000:NEXT

50 POKE36877,0



VIC TIP:

If you turn a particular speaker on it will **STAY ON UNTIL YOU TURN IT OFF**. Example: POKE 36875,200 turns Speaker 2 on with a high-pitched tone. You must POKE 36875,0 to turn the speaker off. Just POKEing the Volume to zero will not turn the speaker off. For example, turn the volume on (POKE36878,15) then POKE a speaker on. Now turn the volume off (POKE36878,0) then turn it on again (POKE36878,15). The tone comes back on again automatically, right? That's because it was never turned off. Just the volume was turned to zero. It's like a radio set on the same station. Whenever you turn the volume up the same station comes on. You have to turn the speaker to a different tone, turn it off by poking zero, or poke it to a number outside of its range to get a "silent" reading (under 128).

MIXING SOUND AND GRAPHICS

Ninety percent of the programs being written with sound or music will combine graphics with sound effects, so here are 3 sample programs which combine graphics and sound:

EXAMPLE 1: CARD GRAPHICS

10 A=97: B=20: C=122: D=115

20 POKE36878,15: S2=36875

30 POKES2,200:PRINTCHR\$(A);

40 FORI=1TO100:NEXTI

50 POKES2,205:PRINTCHR\$(B);

60 FORI=1TO100:NEXTI

70 POKES2,210:PRINTCHR\$(C);

80 FORI=1TO100:NEXTI

90 POKES2,215:PRINTCHR\$(D);

100 FORI=1TO100:NEXTI

110GOTO30

EXAMPLE 2: CALCULATING FORMULA WITH BLIPS

```
10 POKE36878,15:PRINT"ENTER FIRST NUMBER":INPUTA
20 FORX=200TO120STEP-2:POKE36875,X:NEXT
30 PRINT"ENTER SECOND NUMBER":INPUTB
40 FORX=200TO120STEP-2:POKE36875,X:NEXT
50 FORT=1TO200:NEXT
60 PRINTA"MULTIPLIED TIMES" B
  ="A*B:FORX=150TO250STEP2:POKE36875,
X:NEXT:POKE36875,0
70 FORT=1TO5000:NEXT:GOTO10
```

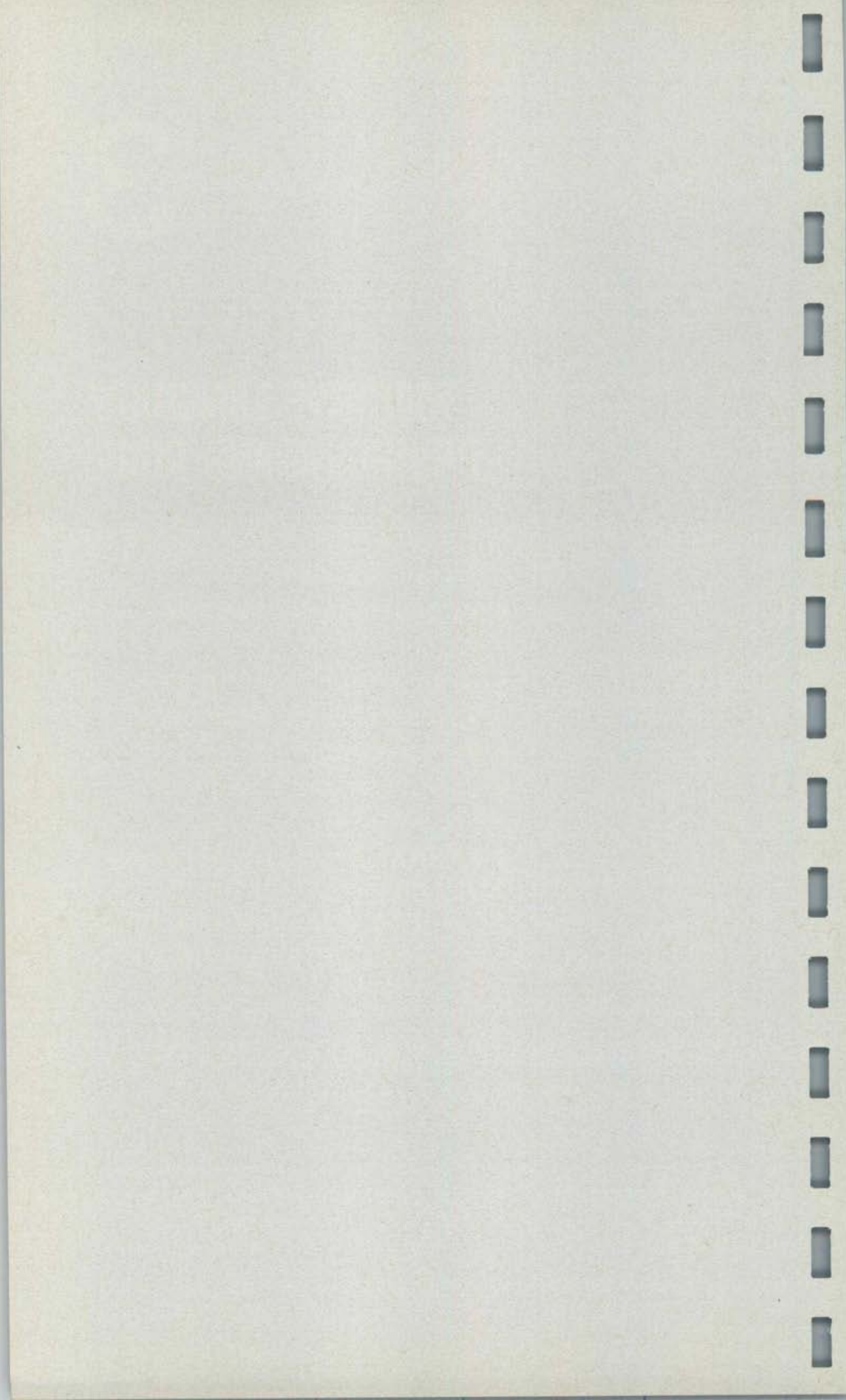
EXAMPLE 3: MUSICAL KEYBOARD

```
10 POKE36878,15:X=128
20 IFX>255THENGOTO10
30 GETA$:IFA$=""THENGOTO30
40 PRINTA$;
50 POKE36876,X
60 X=X+5:GOTO20
```

3

MACHINE LANGUAGE PROGRAMMING GUIDE

- System Overview
- Introduction to Machine Language
- Writing Your First Program
- Special Tips for Beginners
- Memory Maps
- Useful Memory Locations
- The KERNAL
- KERNAL Power Up Activities
- VIC Chips
 - 6560 Video Interface Chip
 - 6522 Versatile Interface Adapter



SYSTEM OVERVIEW

This chapter provides an overall functional description of the VIC 20 and ties hardware and software operations together to give the programmer more of an understanding of the way VIC 20 processes his programs within the system.

A simplified functional block diagram of the computer is shown in Figure 1-1. The major system components include the microprocessor, the program-storage read-only memories (ROMs), the data-storage random-access memories (RAMs), the versatile interface devices (VIAs, 6522), the character generator chip (2332), and the VIC chip which provides video and sound for the display.

The 6502 microprocessor is the most complex device on the electronics printed circuit board. This device is primarily responsible for controlling all computer operations. These operations are controlled by addressing programs in the read-only memory (ROM), and then interpreting and executing these sequential program instructions. The interpretation and execution of instructions are accomplished during the processor's fetch and execute cycles. In the fetch cycle, a program instruction is "fetched" into the processor's instruction register. The program counter (indicates the location of the instruction in ROM) is counted up, ready for the next instruction in sequence to be fetched into the instruction register. In the execute cycle, the processor executes the instruction which performs the operation indicated. Addresses indicating the destination of data being transferred are derived from the instruction, or calculated using program data and data from the internal registers.

These controls exercised by the processor are performed by communicating through the 16-bit address bus, the 8-bit bi-directional data lines, and the write-enable line. The information on the address bus determines the destination of the data being transferred, the bi-directional data bus functions as a path for data transferred into and out of the microprocessor, and the write-enable line determines the direction of the data being transferred.

Consider the microprocessor's inputs and outputs. We can divide these into three groups. Each of these groups forms a "bus" which consists of a set of parallel paths used to transfer binary information between the devices in the system.

The address bus is used to carry the address generated by the microprocessor to the address inputs of the memory and input/output (I/O) devices.

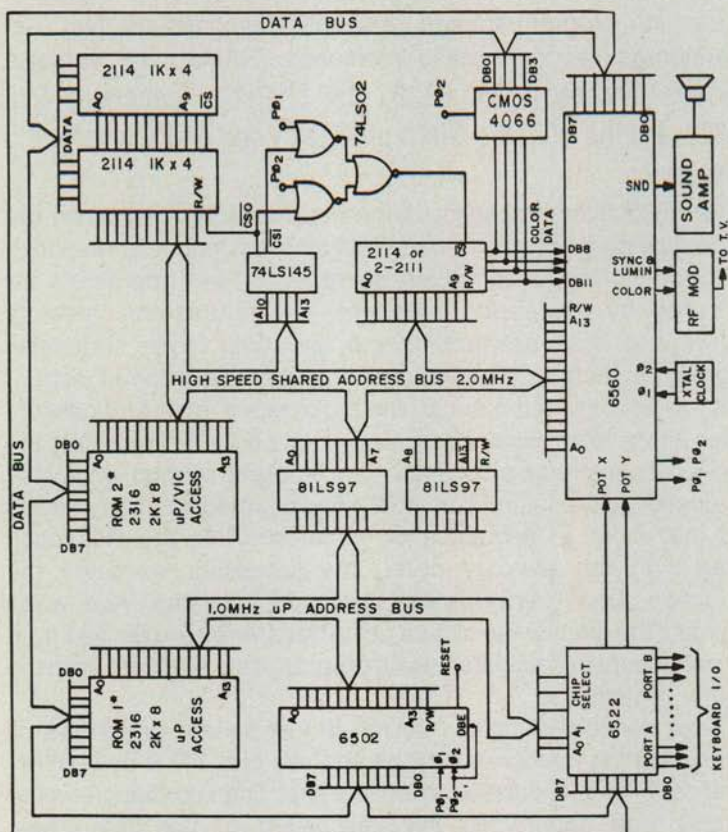


Figure 3-1. VIC system functional block diagram.

The data bus consists of eight bi-directional data lines. During a write operation, these lines transfer data from the processor to a memory location selected by the address lines. During a read operation, data is transferred from memory to the processor along the same lines. The data bus is, therefore, used to carry all data and instructions to and from the processor, memory, and the peripheral devices.

To understand the operation of the control lines which comprise the control bus, we will examine one individually. Since the data bus is bi-directional, the processor must have some method of signalling to memory or I/O to which direction data transfer will take place (whether memory or the I/O is to be read or written to). This function is performed by the R/W (Read/Write) output from the processor. When this line is high, all data transfers will take place from memory to the processors—a read operation. If the R/W line is low, then the processor will write data out to memory.

Other control lines which comprise the control bus are: system clock timing—used to time the operation of the system including data transfers; reset (RST) line—used to initialize the processor when the machine is switched on; and interrupt (IRQ and NMI) lines—used to cause the processor to stop its current program and start a new program at a specified location.

The program memory is the storage for the sequence of BASIC instructions which comprise the system programs. The microprocessor fetches these instructions by placing the appropriate address on the address bus. In response, the memory puts the instruction, in the form of a pattern of 1's and 0's, on the data bus.

The program memory is called a read-only memory because the microprocessor cannot store information into the ROM device. However, by addressing the ROM, the processor can cause the corresponding 8 bits of data to be transferred on the data bus. The ROM is a nonvolatile device, i.e., data is not destroyed when power is disconnected from the system.

The read-write, random-access memory (RAM) provides temporary storage for input data, arithmetic operations, and other data manipulations. Each RAM address corresponds to eight memory cells. However, when power is removed from the system, all RAM-stored data is lost; the RAM is therefore a volatile memory device.

The versatile interface adapters provide interface for the keyboard, user port, control port, and the serial bus. The serial bus provides the communication for peripheral units, such as floppy disk drives, printers and etc. Each port is assigned a unique address to permit communication with the microprocessor (Figure 3-3).

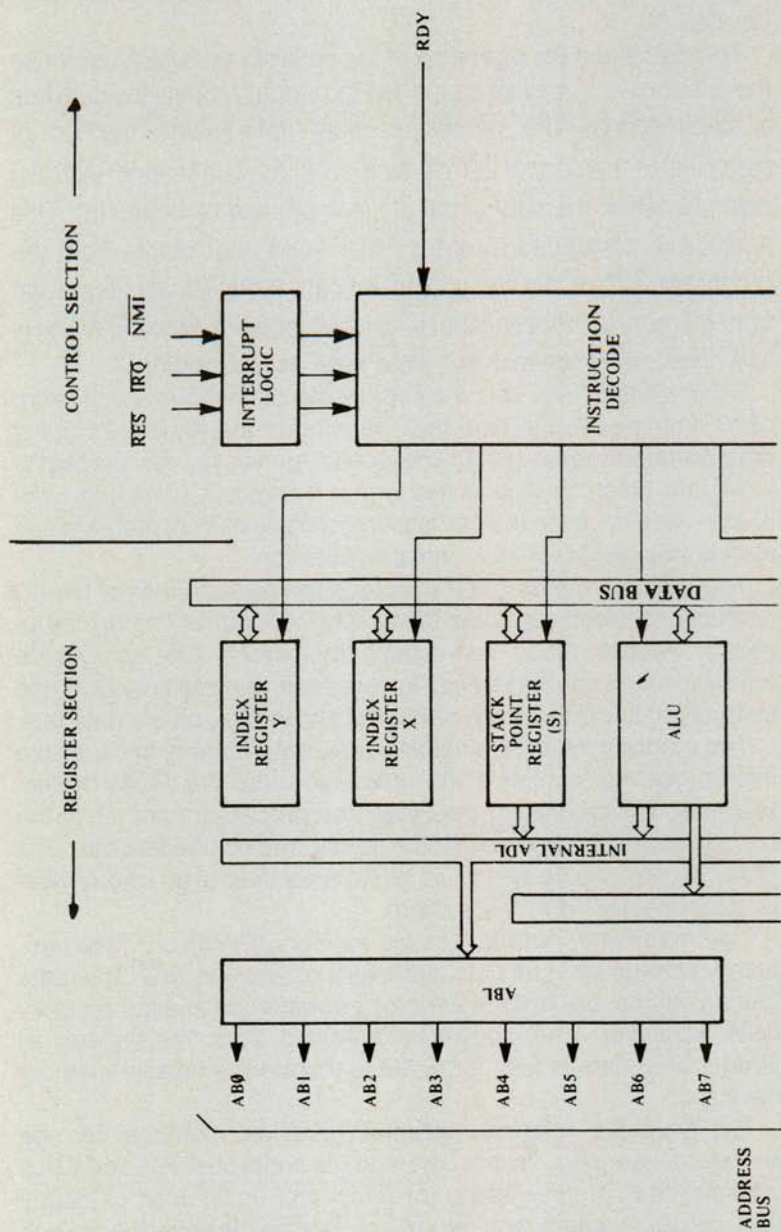
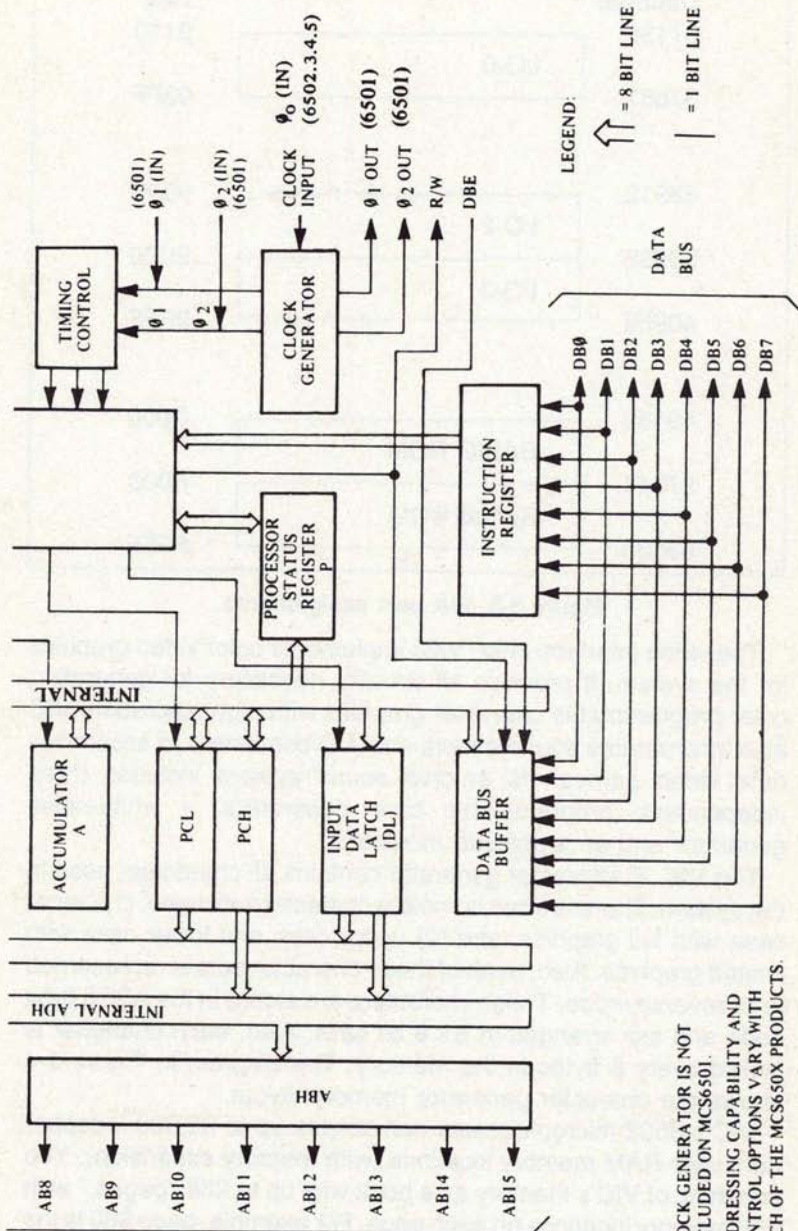


Figure 3-2. The 6502 microprocessor



NOTE: 1. CLOCK GENERATOR IS NOT INCLUDED ON MCS6501.
2. ADDRESSING CAPABILITY AND CONTROL OPTIONS VARY WITH EACH OF THE MCS650X PRODUCTS.

functional block diagram

Decimal		Hex
37136	I/O-0	9110
37887		93FF
38912	I/O-2 I/O-3	9800
39936		9C00
40959		9FFF
49152	BASIC ROM Kernal ROM	C000
57344		E000
65535		FFFF

Figure 3-3. VIA port assignments.

The video interface chip (VIC) implements color video graphics for the system. It provides all circuitry necessary for generating color programmable character graphics with high resolution. VIC also incorporates sound effects and A/D converters to accommodate video games. Its on-chip sound system includes three independent, programmable tone generators, a white-noise generator and an amplitude modulator.

The VIC 20 character generator contains all characters used in the system. There are two complete character sets used: (1) Upper case with full graphics, and (2) upper case and lower case with limited graphics. Also, each of these character sets is represented in its reverse mode. These characters are stored in the ASCII 6-bit code and are arranged in 8×8 bit cells. Also, each character is stored every 8 bytes in the memory. The diagram in Figure 3-4 shows the character generator memory layout.

VIC's 6502 microprocessor can access up to 32,000 independent user-RAM memory locations (with memory expansion). You can think of VIC's memory as a book with up to 256 "pages," with 256 memory locations on each page. For example, page \$80 is the 256 memory locations beginning at location \$8000 and ending at location \$80FF. Since the 6502 uses two 8-bit bytes to form the address of any memory location, you can think of one of the bytes as the page number and the other as the location within the page.

The amount of active RAM may be 3.58K (addresses 4096 to

Decimal		Hex
32768		8000
	Upper case and graphics 1K	
33792		8400
	Upper case and graphics reversed 1K	
33816		8800
	Upper case and lower case 1K	
35840		8C00
	Upper case and lower case reversed 1K	
36863		8FFF

Figure 3-4. Character generator memory layout.

Decimal		Hex
0		0000
	Working Storage RAM 1K	
1024		0400
	Expansion RAM 3K	
4096		1000
	User BASIC Program RAM 4K	
7680		1E00
	Screen RAM	
8192		2000
	Expansion RAM/ROM 8K	
16384		4000
	Expansion RAM/ROM 8K	
24576		6000
	Expansion RAM/ROM 8K	
32768		8000
	Character ROM 4K	
36863		8FFF

Figure 3-5. VIC20 memory locations.

Decimal		Hex
36864		9000
	VIC Chip	
37136		9110
	I/O-0	
37888		9400
	Color RAM	
38912		9800
	I/O-2	
39936		9C00
	I/O-3	
40960		A000
	Expansion ROM 8K	
49152		C000
	BASIC ROM 8K	
57344		E000
	KERNAL ROM 8K	
65535		FFFF

Figure 3-5. (cont).

7679), 6.65K (addresses 1024 to 7679), or a total of 32K by adding 24K more RAM (addresses 8192 to 37267. Addresses 40960 to 49151 are allocated for the expansion of ROM. The first 1K-byte allocation (to 1024) is fixed; the larger the memory size, the more space is available in the user program area.

VIC has three types of memory: random-access memory (RAM), read-only memory (ROM), and input/output locations (I/O). Figure 3-5 shows a typical VIC 20 memory, the different types, and the operations for which they are used.

Each portion of the memory is described in more detail in the following text.

The first 1K-byte of RAM (Addresses 0 – 1023) is allocated to working storage, the stack, and tape buffers. Byte addresses 4096 through 8191 are allocated to screen storage and storage of user programs (Figure 3-6).

Locations 256 through 511 are used for the stack area for BASIC, KERNAL and the microprocessor. The stack begins at location 511 and proceeds downward. Storage is allocated dynamically as needed by BASIC and the hardware. An OUT-OF-MEMORY error occurs if the stack pointer reaches the end of available space in this area.

Locations 512 through 827 are used as additional BASIC and KERNAL working-storage locations.

Locations 828 through 1023 form a tape buffer area for the tape cassette.

Decimal		Hex
0		0000
	BASIC Working Storage	
144		0090
	KERNAL Working Storage	
256		0100
	BASIC & KERNAL Stack	
512		0200
	BASIC & KERNAL Working Storage	
828		033C
	Tape Buffer Working Storage	
1024		0400
	Expansion RAM	
4096		1000
	User BASIC Text	
	
	Variables	
	&	
	Arrays	
	
	Strings	
7680		1E00
	Screen RAM	
8191		1FFF

Figure 3-6. Working storage and user programs .

Locations 4096 through 7679 are used for storage of the user program and variables. The program begins at location 4096 and is stored upward toward the end of memory. Variable storage begins after the end of the program. Array storage begins at the end of variable storage. Strings are stored beginning at the end of memory and working downward. An OUT-OF-MEMORY error occurs if an upgoing pointer meets the downgoing pointer (Figure 3-6).

Addresses 1024 through 4095 are allocated for the expansion of RAM. Addresses 8192 through 32767 are allocated for the expansion of either RAM or ROM, up to 32K-bytes. Addresses 40960 through 49151 are allocated for ROM expansion only (Figure 3-7).

Decimal		Hex
1024	Expansion RAM 3K	0400
4095		0FFF
8192	Expansion RAM/ROM 8K	2000
16384	Expansion RAM/ROM 8K	4000
24576	Expansion RAM/ROM 8K	6000
32767		7FFF
40960	Expansion ROM 8K	A000
49151		BFFF

Figure 3-7. Expansion RAM/ROM.

Locations 37136 through 37887, and 38912 through 40959 are the memory-mapped I/O locations. Locations 49152 through 65535 comprise the BASIC interpreter and KERNAL routines (Figure 3-8).

Decimal		Hex
37136	I/O-0	9110
37887		93FF
38912	I/O-2	9800
39936	I/O-3	9C00
40959		9FFF
49152	BASIC ROM	C000
57344	KERNAL ROM	E000
65535		FFFF

Figure 3-8. BASIC, KERNAL, and I/O locations.

Location 65535 is the end of the VIC memory.

The VIC BASIC interpreter executes a user program by interpreting each source line stored in memory in its compressed form. First, however, a discussion about how the program is stored in memory is necessary.

When a program line is entered from the keyboard, the screen editor takes control, allowing you to edit the line until you press the RETURN key. When the RETURN key is pressed, the BASIC interpreter performs two actions: first, the program line is translated into its compressed form, that is, reserved words and logical-operator keywords are represented by their one-byte tokens; then, the interpreter stores the program line in memory in its ascending line number order. When the RETURN key is pressed, the BASIC interpreter searches memory for the same line number. If there is a line with the same number, it is replaced with the new line. If there is not a line with the same number, the next higher line number is encountered and the interpreter then inserts the new line into memory.

Program lines are stored at the beginning of the user program area of memory which starts at memory location 4096. Variables are stored in memory above the program lines, and arrays are stored above the variables. All three areas begin at lower addresses and build upwards to higher addresses. Strings are stored beginning at the top of memory and work downwards. The BASIC interpreter builds all four areas, moving them as necessary and adjusting pointers for insertions and deletions. Eight pairs of memory locations contain pointers to the division points in the user program area of memory. These pointers are shown in Figure 3-9.

Pointer Address		Typical Values
(2B,2C) Start of Text		4097
	BASIC Statements	
(41,42) DATA Statement Pointer		5879
(2D,2E) Start of variables		5018
	Variables	
(2F,30) End of variables		5144
	Arrays	
(31,32) End of arrays		5303
(33,34) End of strings		7657
	Strings	
(37,38) Top of memory		7679

Figure 3-9. Principal pointers in user program area.

Next, we will discuss the formats in which BASIC statements, variables, arrays, and strings are stored in their respective areas.

The BASIC statement storage table (Figure 3-10) shows the format in which BASIC statements are stored. Memory location 4097 contains a pointer to the beginning of the first BASIC statement. The pointer, like all addresses in the VIC, is stored in low-byte, high-byte order. The pointer is a link to the memory address of the next link. A link address of zero denotes the end of the text; i.e., there are no more links and no more statements. BASIC statements are stored in order of ascending line numbers, even though there are links to the next statement. Links are used to quickly search through line numbers.

The statement line number (stored in low-byte, high-byte order) follows the link address. Line numbers go from 0 to 63999 (stored as 0 and 0, and 255 and 249 respectively).

4097 4098	4099 4101	END
Link	Line# Compressed BASIC Text	0
Link	Line# Compressed BASIC Text	End of statement 0 is flagged by zero byte.
.		
.		
.		
Link 0 0	Line# Compressed BASIC Text (End of text is indicated by two link bytes of zero.)	0

Figure 3-10. BASIC statement storage.

After the line number, the BASIC statement text begins. Reserved words and logical-operator keywords are stored in a compressed format. A one-byte token is used to represent a keyword. All keywords are encoded such that the high-order bit is set to 1. Other elements of the BASIC text are represented by their stored ASCII code. Other elements are comprised of constants, variable and array names, and special symbols other than operators and are coded just as they appear in the original BASIC statement. The BASIC keywords table (Table 3-1) shows the byte codes for all values from 0 to 255 that may appear in the compressed BASIC text. Codes are interpreted according to this table except after an odd number of double quotation marks

enclosing a character string; within a character string the VIC ASCII codes prevail.

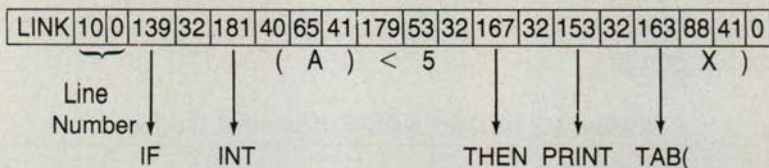
Table 3-1. VIC 20 BASIC Keyword Codes

Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword	Code (decimal)	Character/ Keyword
0	End of line	66	B	133	INPUT	169	STEP
1-31	Unused	67	C	134	DIM	170	+
32	space	68	D	135	READ	171	-
33	!	69	E	136	LET	172	*
34	"	70	F	137	GOTO	173	/
35	#	71	G	138	RUN	174	
36	\$	72	H	139	IF	175	AND
37	%	73	I	140	RESTORE	176	OR
38	&	74	J	141	GOSUB	177	>
39	'	75	K	142	RETURN	178	=
40	(76	L	143	REM	179	<
41)	77	M	144	STOP	180	SGN
42	*	78	N	145	ON	181	INT
43	+	79	O	146	WAIT	182	ABS
44	,	80	P	147	LOAD	183	USR
45	-	81	Q	148	SAVE	184	FRE
46	.	82	R	149	VERIFY	185	POS
47	/	83	S	150	DEF	186	SQR
48	0	84	T	151	POKE	187	RND
49	1	85	U	152	PRINT#	188	LOG
50	2	86	V	153	PRINT	189	EXP
51	3	87	W	154	CONT	190	COS
52	4	88	X	155	LIST	191	SIN
53	5	89	Y	156	CLR	192	TAN
54	6	90	Z	157	CMD	193	ATN
55	7	91	[158	SYS	194	PEEK
56	8	92	\	159	OPEN	195	LEN
57	9	93]	160	CLOSE	196	STR\$
58	:	94	↑	161	GET	197	VAL
59	:	95	←	162	NEW	198	ASC
60	<	96-127	Unused	163	TAB(199	CHR\$
61	=	128	END	164	TO	200	LEFT\$
62	>	129	FOR	165	FN	201	RIGHT\$
63	?	130	NEXT	166	SPC(202	MID\$
64	@	131	DATA	167	THEN	203-254	Unused
65	A	132	INPUT#	168	NOT	255	π

Note that the left parenthesis is stored as part of the one-byte token for functions TAB and SPC; however, the other functions use a separate byte for this symbol. For example, the line

10 IF INT(A)<5 THEN PRINT TAB(X)

would be coded as the following bytes (in decimal):



The operators (+ - * / < = >) as well as the words AND, OR, and NOT are given keyword codes (high-order bit set) since they "drive" the BASIC interpreter just as reserved words do (e.g., 179 for <). The standard ASCII codes for these symbols (e.g., 60 for <) appear only in the text of a string.

Spaces in the source line are stored except for the space between the line number and first keyword. This space is supplied on LISTing when a stored statement is expanded to its original form. You can conserve memory storage space by eliminating blanks (but this makes the program harder to read). You can also conserve space by putting more than one statement on a line, since the five bytes of link, line number, and 0 end byte are stored only once.

The size of each statement is variable and is terminated by a byte of zero to indicate the end of the statement. (An ASCII zero anywhere within the text is stored as 48.) Zero-byte flags are used by the BASIC interpreter in executing a program when it goes through the compressed BASIC text from left to right picking out keywords and performing the indicated operations. A zero byte indicates the end of the statement; the next four bytes are the link to the line number of the next statement. Instead of searching through the text and using 0 byte indicators to locate the next statement, links are used when searching the statements for their line numbers. Three consecutive bytes of zero (the last statement's 0 byte followed by two zero link bytes) flag the end of text when executing the program.

A program stored onto cassette tape is in the same format as shown in Figure 3-10 for memory storage. Therefore, it is basically "dumped" onto tape in a continuous block, including link addresses and 0 end bytes.

The use of tokens in place of keywords is not unique to the VIC, but there is no standard coding from one interpreter to another. Thus, a BASIC source program SAVED on tape by VIC BASIC is not compatible with other BASICs, nor can BASIC programs generated on other (non-CBM) machines normally be loaded by the VIC BASIC interpreter.

INTRODUCTION TO MACHINE LANGUAGE

WHAT IS MACHINE LANGUAGE?

At the heart of every microcomputer, there is a central microprocessor, a very special microchip which is the "brain" of the computer. The VIC 20's microprocessor is the 6502 chip. Every microprocessor understands its own language of instructions, and these instructions are called the machine language instructions of that chip. To put it more precisely, machine language is the **ONLY** programming language that your VIC 20 really understands. It is the native language of the machine.

If machine language is the only language that the VIC 20 understands, then how does it understand the VIC BASIC programming language? If VIC BASIC is not the machine language of the VIC 20, what makes the VIC 20 understand VIC BASIC instructions such as PRINT and GOTO?

To answer this question, we must first see what happens to your VIC 20 when you turn it on. How does your computer know what to do when it is first turned on? Well, apart from the microprocessor which is the brain of the VIC 20, there is a huge machine language program which is "burnt" into a special type of memory called ROM that cannot be changed, and does not get erased when the VIC 20 is turned off, unlike a program that you put into the VIC's RAM. This huge program is in two parts, one taking care of the BASIC language, and the other called the "operating system."

The operating system is in charge of "organizing" all the memory in your machine for various tasks, looks at what characters you type on the keyboard and puts them onto the screen, and a whole number of other functions. The operating system can be thought of as the "intelligence and personality" of the VIC 20 (or any computer for that matter). So when you turn on your VIC 20, the operating system takes control of your machine, and after it has done its housework, it then says:

READY.

*

The operating system of the VIC 20 then allows you to type on the keyboard, and use the built-in "screen editor" on the VIC 20. The screen editor allows you to move the cursor, DELeTe, INSert, etc., and is, in fact, only one part of the operating system that is built-in for your convenience.

All of the commands that are available in VIC BASIC are simply recognized by another huge machine language program built into your VIC 20. This huge program "RUN" 's the appropriate piece of machine language depending on which VIC BASIC command is being executed. This program is called the "BASIC interpreter," because it interprets each command, one by one, unless it encounters a command it does not understand, and then the familiar message appears:

?SYNTAX

ERROR

READY.

*

WHAT DOES MACHINE CODE LOOK LIKE?

You should be familiar with the PEEK, and POKE commands in the CBM BASIC language for changing memory locations. You will probably have used them for graphics on the screen, and for sound effects. The memory locations will have been 36874, 36875, 36876, 36877, 36878 for sound effects. This memory location number is known as the "address" of a memory location. If you can imagine the memory in the VIC 20 as a street of houses, the number on the door is, of course, the address. Now we will look at which parts of the street are used for which purpose.

SIMPLE MEMORY MAP OF THE VIC 20

Address	Description
0	Start of memory.
to	Memory used by BASIC and the operating system.
1023	
1024	
to	This is a gap in memory for a 3K memory expansion module
4095	
4096	
	This is YOUR memory. This is where your BASIC or machine language programs, or both, are stored.
to	This is also where the screen memory would begin on a VIC 20 that has at least 8K of RAM expansion memory, in which case the screen RAM that follows would become user memory space,
7679	continuing up to the top of the expansion memory.

7680	
to	This is the screen memory.
8185	
to	This is a gap in memory for memory expansion.
32768	
to	Character representations.
36863	
36864	
to	The VIC chip registers.
36879	
37136	
to	Input and output chip registers.
37167	
37888	
to	Character color control table in expanded VIC 20
38399	
38400	
to	Character color control table.
38911	
38912	
to	Unused.
40959	
40960	
to	Expansion ROM.
49151	
49152	
to	8K VIC BASIC Interpreter.
57343	
57344	
to	8K VIC KERNAL OPERATING SYSTEM
65535	

Don't worry if you don't understand what the description of each part of memory means. This will become clear from other parts of this manual.

Machine language programs consist of instructions which may or may not have operands (parameters) associated with them. Each instruction takes up one memory location, and any operand will be contained in one or two locations following the instruction.

In your BASIC programs, words like PRINT, and GOTO do, in fact, only take up one memory location, rather than one for each character of the word. The contents of the location that represents a particular BASIC keyword is called a "token." In machine language,

there are different tokens for different instructions, which also take up just one byte (memory location = byte).

Machine language instructions are very simple, i.e., each individual instruction cannot achieve a great deal. Machine language instructions either change the contents of a memory location, or change one of the internal registers (special storage locations) inside the microprocessor. The internal registers form the very basis of machine language.

REGISTERS INSIDE THE 6502 MICROPROCESSOR

THE ACCUMULATOR—This is THE most important register in the microprocessor. Various machine language instructions allow you to copy the contents of a memory location into the accumulator, or copy the contents of the accumulator into a memory location, or modify the contents of the accumulator or some other register directly, without affecting any memory. Also, the accumulator is the only register that has instructions to perform math on it.

THE X INDEX REGISTER—There are instructions to do nearly all of the transformations you can do to the accumulator, and other instructions to do things that only the X register can do. Again, various machine language instructions allow you to copy the contents of a memory location into the X register, or copy the contents of the X register into a memory location, or modify the contents of the X, or some other register directly, without affecting any memory.

THE Y INDEX REGISTER—There are instructions to do nearly all of the transformations you can do to the accumulator, and the X register, and other instructions to do things that only the Y register can do. Again, various machine language instructions allow you to copy the contents of a memory location into the Y register, or copy the contents of the Y register into a memory location, or modify the contents of the Y, or some other register directly, without affecting any memory.

THE STATUS REGISTER—This register consists of eight “flags” (a flag = something that indicates that something has, or has not, occurred).

THE PROGRAM COUNTER—This contains the address of the current machine language instruction being executed. Since the

operating system is always "RUN"ning in the VIC 20 (or, for that matter, any computer), the program counter is always changing. It could only be stopped by halting the microprocessor in some way.

The Stack Pointer—This register contains the location of the first empty place on the stack. The stack is used for temporary storage by machine language programs, and by the computer.

THE TOOLS AVAILABLE; GETTING READY...

How Can You Write Machine Language Programs?

Since machine language programs reside in memory, and there is no facility in your VIC 20 for writing and editing machine language programs, you must use either a program to do this, or write for yourself a BASIC program that "allows" you to write machine language.

Most commonly used to write machine language programs are "assemblers." These packages allow you to write machine language instructions in a standardized "mnemonic" format, which makes the machine language program a good deal more readable than a stream of numbers. To recap: A program that allows you to write machine language programs in mnemonic format is called an "assembler," and also, a program that displays a machine language program in mnemonic format is called a "disassembler." Available for your VIC 20 is a machine language monitor cartridge (with assembler/disassembler, etc.) made by Commodore.

VICMon

The VICMon cartridge available from your local dealer is a program that allows you to escape from the world of VIC BASIC, into the land of machine language. It can display the contents of the internal registers in the 6502 microprocessor, and it allows you to display portions of memory, and change them on the screen, using the screen editor. It also has a built-in assembler and disassembler, and many other features that allow you to write and edit machine language programs easily.

You don't HAVE to use an assembler to write machine language, but the task is considerably easier with it. If you wish to write machine language programs, it is advised strongly that you buy an assembler of some sort. Without an assembler you will probably have to "POKE" the machine language program into memory, which, if you value your sanity, is totally inadvisable. This manual

will give examples in the format that VICMon uses from now on. Nearly all assembler formats are the same, therefore the machine language examples shown will almost certainly be compatible with any assembler other than the one incorporated in VICMon.

Hexadecimal Notation

This is a notation which most machine language programmers refer to when referring to a number or address in a machine language program.

Some assemblers let you refer to addresses and numbers in decimal (base 10), binary (base 2), or even octal (base 8) as well as hexadecimal (or just "hex" as most people say). These assemblers do the conversions for you.

Hexadecimal will probably seem a little hard to grasp at first, but like most things it doesn't take long (with practice) to master it.

By looking at decimal (base 10) numbers, you will see that each digit in that number ranges between zero and a number equal to the base less one, i.e., < 10 . THIS IS TRUE OF ALL NUMBER BASES. Binary (base 2) numbers have digits ranging from zero to one (which is one less than the base). Similarly hexadecimal numbers should have digits ranging from zero to fifteen, but we do not have any single digit figures for the numbers ten to fifteen, so the first six letters of the alphabet are used instead:

DECIMAL		HEXADECIMAL		BINARY
0	—	0	—	00000000
1	—	1		00000001
2	—	2	—	00000010
3	—	3		00000011
4	—	4	—	00000100
5	—	5		00000101
6	—	6	—	00000110
7	—	7		00000111
8	—	8	—	00001000
9	—	9		00001001
10	—	A	—	00001010
11	—	B		00001011
12	—	C	—	00001100
13	—	D		00001101
14	—	E	—	00001110
15	—	F		00001111
16	—	10	—	00010000
etc.				

If that's confusing, let's try to look at it another way:

Example of how a base 10 (decimal number) is constructed.

Base raised by 3 2 1 0
increasing powers.. 10 10 10 10

Equals:.....1000 100 10 1

Consider 4569 (base10) $4\ 5\ 6\ 9 = 4 \times 1000 + 5 \times 100 + 6 \times 10 + 9$

Example of how a base 16 (hexadecimal number) is constructed.

Base raised by 3 2 1 0
increasing powers.. 16 16 16 16

Equals:.....4096 256 16 1

Consider 11D9 (base16) $1\ 1\ D\ 9 = 1 \times 4096 + 1 \times 256 + 13 \times 16 + 9$

Therefore $4569(\text{Base}10) = 11D9(\text{Base}16)$

The range for addressable memory locations is 0 – 65535 (as was stated earlier). This range is therefore 0 – FFFF in hexadecimal notation.

Usually hexadecimal numbers are prefixed with a dollar sign, to distinguish them from decimal numbers. Let's look at some "hex" numbers, using VICMon by displaying the contents of some memory. VICMon shows you:

B*

PC SR AC XR YR SP

.; 0401 32 04 5E 00 F6 (these may be different)

Then if you type in:

.M 0000 0020 (and press RETURN).

You will see rows of 6 hex numbers. The first 4 digit one is the address of the first byte of memory being shown in that row, and the other five numbers are the actual contents of the memory locations beginning at that start address.

You should endeavor to learn to "think" in hexadecimal. This is not difficult, since there is no need to think in decimal. For example, if it is said that a particular value is stored at \$14ED instead of 5357, this shouldn't cause any headaches.

YOUR FIRST MACHINE LANGUAGE INSTRUCTION

"LDA"—Load the Accumulator

In 6502 assembly language, mnemonics are always three characters. LDA represents "load accumulator with. . .", and what the accumulator should be loaded with is decided by the parameter(s) associated with that instruction. The assembler knows which token is represented by each mnemonic, and when it "assembles" an instruction, it simply puts into memory (at whatever address has been specified), the token and what parameters are given. Some assemblers give error messages, or warnings when the user has tried to assemble something that either the assembler or the 6502 microprocessor cannot do.

If we put a "#" symbol in front of the parameter associated with the instruction, this means that we wish the register specified in the instruction to be loaded with the "value" after the "#". For example:—

LDA #\$05

This instruction will put \$05 (decimal 5) into the accumulator register. The assembler will put into the specified address for this instruction, \$A9 (which is the token for this particular instruction, in this mode), and it will put \$05 into the next location after the location containing the instruction (\$A9).

If the parameter to be used by an instruction has a "#" before it, i.e., the parameter is a "value," rather than the contents of a memory location, or another register, the instruction is said to be in the "immediate" mode. To put this into perspective, let us compare this with another mode.

If we want to put the contents of memory location \$102E into the accumulator, we are using the "absolute" mode of instruction:

LDA \$102E

The assembler can distinguish between the two different modes because the latter does not have a "#" before the parameter. The 6502 microprocessor can distinguish between the *immediate mode* and the *absolute mode* of the LDA instruction because they have slightly different tokens. LDA (immediate) has \$A9 (as stated previously), and LDA (absolute) has \$AD.

The mnemonic representing an instruction usually implies what it does. For instance, if we consider another instruction, "LDX," what do you think this does?

If you said "load the X register with. . .", go to the top of the class.

If you didn't, then don't worry; learning machine language does take patience, and cannot be accomplished in a day.

The various internal registers can be thought of as special memory locations, because they too can hold one byte of information. It is not necessary for us to explain the binary numbering system (base 2) since it follows the same rules as outlined for hexadecimal and decimal outlined previously, but one "bit" is one binary digit and eight bits make up one byte.

The maximum number that can be contained in a byte is the largest number that an eight digit binary number can be. This number is 11111111 (binary), which equals \$FF (hexadecimal), which equals 255 (decimal). You have probably wondered why only numbers from zero to two hundred and fifty-five could be put into a memory location. If you try POKE 7680,260 (which is a BASIC statement that "says":—"Put the number two hundred and sixty into memory location seven thousand, six hundred and eighty." The BASIC interpreter knows that only numbers 0 – 255 can be put in a memory location, and your VIC 20 will reply with:

```
?ILLEGAL QUANTITY
ERROR
READY.
```

*

If the limit of one byte is \$FF (hex), how is the address parameter in the absolute instruction "LDA \$102E" expressed in memory?

Well, it is expressed in two bytes (it won't fit into one, of course). The lower (rightmost) two digits of the hexadecimal address form the "low byte" of the address, and the upper (leftmost) two digits form the "high byte."

The 6502 requires any address to be specified with its low byte first, and then the high byte. This means that the instruction "LDA \$102E" is represented in memory by the three consecutive values:

\$AD, \$2E, \$10

We need to know one more instruction before we can write our first program. That instruction is "BRK." For a full explanation of this instruction, refer to M.O.S 6502 Programming Manual. You can think of it as the "END" instruction in machine language.

If we write a program with VICMon and put the BRK instruction at the end, the program will return to VICMon when it is finished. This might not happen if there is a mistake in your program, or if the BRK instruction is never reached (just like an "END" statement in BASIC may never get executed, and thus if the VIC 20 didn't have a STOP key, you wouldn't be able to abort your BASIC programs!)

WRITING YOUR FIRST PROGRAM

If you have used the POKE statement in BASIC to put characters onto the screen, you will be aware that the character codes for POKEing are different to CBM ASCII character values. For example, if you enter:

```
PRINT ASC("A") (and press <RETURN> )
```

The VIC 20 will respond with:

```
65  
READY.  
*
```

However, to put an "A" onto the screen by POKEing, the code is 1. Since the screen memory starts at 7680 (decimal), or 4096 if you have 8K or more of expansion memory, by entering:

```
<CLR> (To clear the screen)  
POKE 7680,1 (and <RETURN> ) (NOTE: POKE 4096,1 on a  
VIC 20  
with 8K or more of  
expansion memory)
```

The "P" in the POKE statement should now be an "A." We will now do this in machine language. Type the following in VICMon:

(Your cursor should be flashing alongside a "." right now.)

```
.A 1400 LDA #$01 (and press <RETURN> )
```

The VIC will prompt you with:

```
.A 1402 *
```

Type:

```
.A 1402 STA $1E00 (or STA $1000 on a VIC 20  
with 8K or more of expansion memory)
```

The STA instruction stores the contents of the accumulator in a specified memory location. The VIC will now prompt you with:

```
.A 1405 *
```

Now enter:

```
.A 1405 BRK
```

Clear the screen, and type:

```
G 1400
```

The G should turn into an "A" if you have done everything correctly. You have now written your first machine language program! Its purpose is to store one character, the letter A, in the first byte of screen memory.

ADDRESSING MODES

ZERO PAGE

As shown earlier, absolute addresses are expressed in terms of a high order and a low order byte. The high order byte is often referred to as the page of memory. For example, the address \$1637 is in page \$16 (22), and \$0277 is in page \$02 (2). There is, however, a special mode of addressing known as "zero page" addressing and it is, as the name implies, associated with the addressing of memory locations in page zero. These addresses have a high order byte of zero. The zero page mode of addressing only expects one byte to describe the address, rather than two when using an absolute address, which saves speed and time. This mode tells the microprocessor to assume that the high order address is zero. Therefore zero page addressing can reference memory locations whose addresses are between \$0000, and \$00FF.

THE STACK

The 6502 microprocessor (like almost all others) has what is known as a "stack." This is used both by the programmer and the microprocessor to temporarily remember things, and to remember the order of events. The GOSUB statement in BASIC, which allows the programmer to call a "subroutine," must remember where it is being called from. When the RETURN statement is executed in the subroutine, the BASIC interpreter "knows" where to go back in order to continue executing. When a GOSUB statement is encountered in a program by the BASIC interpreter, the BASIC interpreter "pushes" its current position onto the stack before going to do the subroutine, and when a RETURN is executed, the interpreter "pulls" from the stack the information that tells it where it was before the subroutine call was made, so that it may continue as if nothing had happened. The interpreter uses instructions like PHA which will push the contents of the accumulator onto the stack, and PLA (the inverse) which will pull a value off the stack into the accumulator. The status register can also be pushed and pulled with the PHP, and PLP respectively.

The stack is 256 bytes long, and is located in page one of memory. It is therefore from \$0100 to \$01FF. It is organized

backwards in memory, i.e., the first position in the stack is at \$01FF, and the last is at \$0100. Another register in the 6502 microprocessor that hasn't been mentioned yet is called the "stack pointer," and it always points at the next available location in the stack. When something is pushed onto the stack, it is placed where the stack pointer points to, and the stack pointer is moved down to the next position (decremented). When something is pulled off the stack, the stack pointer is incremented, and the byte pointed to by the stack pointer (at \$0100 offset by the contents of the stack pointer) is placed into the specified register.

Up to this point, we have covered immediate, zero page, and absolute mode instructions. We have also covered (but have not stated) the "implied" mode, which means that the instruction itself tells what registers/flags/memory the instruction is referring to. The examples we have seen are PHA, PLA, PHP, and PLP, which refer to stack processing and the accumulator and status registers.

The X register will be referred to as X from now on, and similarly with A — accumulator, Y — Y index register, S — stack pointer, and P — processor status).

INDEXING

Indexing plays an extremely important part in the running of the 6502 microprocessor. It can be defined as "creating an actual address from a base address plus the contents of either the X or Y index registers."

For example, if X contains \$05, and the microprocessor executes an LDA instruction in the "absolute X indexed mode" with base address, e.g., \$9000, then the actual location that is loaded into the A register is $\$9000 + \$05 = \$9005$. The mnemonic format of an absolute indexed instruction is the same as an absolute instruction except a "X" or "Y" denoting the index is added to the address, e.g.:

```
LDA $9000,X
```

INDIRECT INDEXED ADDRESSING

This mode allows the program to choose a memory location from 256 adjacent locations. The address of the lowest location is stored in zero page, and the value in the Y register is added to that address to choose the final address.

For example, we will place a \$45 in location \$01, and a \$1E in location \$02. We will use the instruction to load the accumulator in the indirect indexed mode, specifying zero page address \$01 as the

location where the address to be used is held. Then the actual address will be comprised of:

low address byte = contents of \$01 = \$45
high address byte = contents of \$02 = \$1E
Y register = \$10

The actual address = $\$1E45 + Y = \$1E55$

If you think of indexed addressing like delivering junk mail through a post office, here is the principle for *indirect indexed addressing*:

We will deliver the letters to all the houses on the block starting at \$1E00 Memory St. and continuing for 256 houses. Here is the equivalent program for VICMon:

- | | |
|-----------------------|---|
| • A 1200 LDA #\$00 | load low order actual base address |
| • A 1202 STA \$01 | set the low byte of the first indirect address |
| • A 1204 STA \$FE | set the low byte of the second address |
| • A 1206 LDA #\$1E | load high order indirect address |
| • A 1208 STA \$02 | set the high byte of the first indirect address |
| • A 120A LDA #\$96 | load the second address's high byte |
| • A 120C STA \$FF | set the high byte of the second address |
| • A 120E LDY #\$00 | set the indirect index (Y) |
| • A 1210 LDA #\$66 | 66 is the value of our "letter" to the first "block" |
| • A 1212 STA (\$01),Y | store the "letter" in the house on the first "block" |
| • A 1214 LDA #\$0A | 0A is the value of our second "letter" |
| • A 1216 STA (\$FE),Y | store the "letter" in the house on the second "block" |
| • A 1218 INY | add 1 to index |
| • A 1219 BNE \$1210 | branch back & send next letter |
| • A 121B BRK | return to VICMon when done |
| • G 1200 | sends the "letter"—fills the top of the screen with blue & red lines! |

INDEXED INDIRECT ADDRESSING

This mode allows the program to choose an address from a table in page zero. Since page zero space is limited to 256 bytes, this is a mode that isn't used too often.

This mode only works with the X register. It is like indirect indexed, except that the zero page location is indexed, rather than an address stored in zero page. Therefore, the address stored in

page zero is the actual address because the index has already been used in the indirection.

Let us fill location \$05 with \$45, and location \$06 with \$1E. If the instruction to load the accumulator in the indexed indirect mode is executed and the specified zero page address is \$01, then the actual address will be comprised of:

low order = contents of ($\$01 + X$)
high order = contents of ($\$02 + X$)
X register = \$04

Thus the actual address will be in $= \$01 + X = \05

Therefore, the actual address will be the indirect address contained in \$05 and \$06 which is \$1E45

This is like sending a mailing to a specific list of addresses. We will store a list in zero page, and send the "letter" only to those in the list. Suppose the list of addresses starts at \$00. Here is a program to send a "letter" to one of the addresses:

LDA #\$00	—load low order actual base address
STA \$06	—set the low byte of the indirect address
LDA #\$16	—load high order indirect address
STA \$07	—set the high byte of the indirect address
LDX #\$06	—set the indirect index (X)
LDA (\$00,X)	—load indirectly indexed by X.

BRANCHES AND TESTING

Another very important principle in machine language is the ability to test, and detect certain conditions, in a similar fashion to the "IF. . . THEN" structure in VIC BASIC.

The various "flags" in the status register are affected by different instructions in different ways. For example, there is a flag that is set when an instruction has caused a zero result, and is reset when a result is not zero.

LDA #\$00

This instruction will cause "the zero result flag" to be set, because the instruction has resulted in the accumulator containing a zero.

There is a set of instructions that will, given a particular condition, "branch" to another part of the program. An example of a branch instruction is "BEQ", which means "branch if result equal to zero." The branch instructions will "branch" if the condition is true, and if not, the program will continue onto the next instruction, as if nothing had occurred. The branch instructions branch not by the result of

the previous instruction(s), but by internally examining the status register.

As was just mentioned, there is a "zero result" flag in the status register. The "BEQ" instruction branches if the "zero result" flag (known as "Z") is set. Every branch instruction has an opposite branch instruction. The BEQ instruction has an opposite instruction "BNE" ("branch on result NOT equal to zero," i.e., "Z" not set).

The index registers have a number of associated instructions which modify their contents. For example, the "INX" instruction will "increment the X index register." If the X register contained \$FF before it was incremented (the maximum number the X register can contain), it will "wrap around" back to zero. If we wanted a program to continue to do something until we had performed the increment of the X index that pushed it around to zero, we could use the BNE instruction to continue "looping" around, until X became zero.

Apart from INX, there is "DEX", which will decrement the X index register. If it is zero, it will wrap around to \$FF. Similarly, there are "INY" and "DEY" for the Y index register.

But what if a program didn't want to wait until X or Y had reached (or not reached) zero? Well there are comparison instructions, "CPX" and "CPY", which allow the machine language programmer to test the index registers with specified values, or even the contents of memory locations. If we wanted to see if the X register contained \$40, we would use the instruction:

CPX #\$40	compare X with the "value" \$40.
BEQ (some other part of the program)	branch to somewhere else in the program, if this condition is "true."

The compare and branch instructions play a major part in any machine language program.

The operand specified in a branch instruction when using VICMon is the address of the part of the program the branch should go to, if taken. However, the operand is only, in fact, an "offset" from where the program currently is, to the address specified. This offset is just one byte, and therefore the range that a branch instruction can branch to is limited from 128 bytes backward, to 127 bytes forward; this is a total range of 255 bytes, which is, of course, the maximum range of values one byte can contain. VICMon will tell you if you branch out of range, by refusing to "assemble" that instruction. It is unlikely that you will be doing such huge branches for quite a while anyway. For nearly all situations this is adequate anyway. The branch is a "quick" instruction by machine language standards because of this "offset" principle as opposed to an

absolute address. VICMon allows you to type in an absolute address, and it calculates the correct offset. This is just one of the "comforts" of using an assembler.

Subroutines

In machine language (in the same way as using BASIC), you can call subroutines. The instruction to call a subroutine is "JSR" (jump to subroutine), followed by the specified absolute address.

Incorporated in the operating system is a machine language subroutine that will PRINT a character to the screen. The CBM ASCII code of the character should be in the accumulator before calling the subroutine. The address of this subroutine is \$FFD2.

Therefore, to print "HI" to the screen, the following program should be entered:

- A 1400 LDA #\$48 load the CBM ASCII code of "H"
- A 1402 JSR \$FFD2 print it
- A 1405 LDA #\$49 load the CBM ASCII code of "I"
- A 1407 JSR \$FFD2 print that too
- A 140A LDA #\$0D print a carriage return as well
- A 140C JSR \$FFD2
- A 140F BRK return to VICMon.
- G 1400 will print "HI" and return to VICMon

The "PRINT a character" routine we have just used is part of the KERNAL "jump table." The instruction similar to GOTO in BASIC, is "JMP," which means "jump to the specified absolute address." The KERNAL is a long list of "standardized" subroutines that control ALL input and output of the VIC 20. Each entry in the KERNAL JMP's to a subroutine in the operating system. This "jump table" resides at \$FF84 to \$FFF5 in the operating system. A full explanation of the KERNAL is in the "KERNAL REFERENCE SECTION" in this manual, but certain routines will be used here to show how easy, and effective, the KERNAL is.

We will now use these new principles in another program which will help you to put these instructions into context:

This program will display the alphabet using a KERNAL routine.

The only new instruction introduced here is TXA "transfer the contents of the X index register, into the accumulator."

- A 1400 LDX #\$41 X = CBM ASCII of "A".
- A 1402 TXA A = X.
- A 1403 JSR \$FFD2 print character.
- A 1406 INX bump count.
- A 1407 CPX #\$51 have we gone past "Z" ?

- A 1409 BNE \$1402 no—go back and do more.
- A 140B BRK yes—return to VICMon.

To see the VIC print the alphabet, type the familiar command:

.G 1400

The comments that are beside the program explain the program flow, and logic. If you are writing a program, write it on paper first, and test it in small parts if possible.

MCS6501-MCS6505 MICROPROCESSOR

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location

INSTRUCTION SET – ALPHABETIC SEQUENCE

JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

The following notation applies to this summary:

A	Accumulator
X, Y	Index Registers
M	Memory
P	Processor Status Register
S	Stack Pointer
✓	Change
—	No Change
+	Add
∧	Logical AND
—	Subtract
⊕	Logical Exclusive Or
↑	Transfer from Stack
↓	Transfer to Stack
→	Transfer to
←	Transfer from
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	OPERAND
#	IMMEDIATE ADDRESSING MODE

Note: At the top of each table is located in parentheses a reference number (Ref: XX) which directs the user to that Section in the MCS6500 Microcomputer Family Programming Manual in which the instruction is defined and discussed.

ADC*Add memory to accumulator with carry***ADC**Operation: $A + M + C \rightarrow A, C$

N Z C I D V

(Ref: 2.2.1)

✓ ✓ ✓ — — ✓

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ADC # Oper	69	2	2
Zero Page	ADC Oper	65	2	3
Zero Page, X	ADC Oper, X	75	2	4
Absolute	ADC Oper	6D	3	4
Absolute, X	ADC Oper, X	7D	3	4*
Absolute, Y	ADC Oper, Y	79	3	4*
(Indirect, X)	ADC (Oper, X)	61	2	6
(Indirect), Y	ADC (Oper), Y	71	2	5*

* Add 1 if page boundary is crossed.

AND*"AND" memory with accumulator***AND**

Logical AND to the accumulator

Operation: $A \wedge M \rightarrow A$

N Z C I D V

(Ref: 2.2.3.0)

✓ ✓ — — —

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	AND # Oper	29	2	2
Zero Page	AND Oper	25	2	3
Zero Page, X	AND Oper, X	35	2	4
Absolute	AND Oper	2D	3	4
Absolute, X	AND Oper, X	3D	3	4*
Absolute, Y	AND Oper, Y	39	3	4*
(Indirect, X)	AND (Oper, X)	21	2	6
(Indirect), Y	AND (Oper), Y	31	2	5

* Add 1 if page boundary is crossed.

ASL**ASL Shift Left One Bit (Memory or Accumulator)****ASL**Operation: C +

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

 + 0

N Z C I D V

✓ / ✓ - - -

(Ref: 10.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	0A	1	2
Zero Page	ASL Oper	06	2	5
Zero Page, X	ASL Oper, X	16	2	6
Absolute	ASL Oper	0E	3	6
Absolute, X	ASL Oper, X	1E	3	7

BCC**BCC Branch on Carry Clear****BCC**

Operation: Branch on C = 0

N Z C I D V

- - - - -

(Ref: 4.1.1.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC Oper	90	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BCS**BCS Branch on carry set****BCS**

Operation: Branch on C = 1

N Z C I D V

- - - - -

(Ref: 4.1.1.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS Oper	B0	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BEQ**BEQ** Branch on result zero**BEQ**Operation: Branch on $Z = 1$

N Z C I D V

(Ref: 4.1.1.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ Oper	F0	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BIT**BIT** Test bits in memory with accumulator**BIT**Operation: $A \wedge M, M_7 \rightarrow N, M_6 \rightarrow V$

Bit 6 and 7 are transferred to the status register.

N Z C I D V

If the result of $A \wedge M$ is zero then $Z = 1$, otherwise $M_7 \vee \dots \vee M_6$ $Z = 0$

(Ref: 4.2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT Oper	24	2	3
Absolute	BIT Oper	2C	3	4

BMI**BMI** Branch on result minus**BMI**Operation: Branch on $N = 1$

N Z C I D V

(Ref: 4.1.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI Oper	30	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BNE**BNE** Branch on result not zero**BNE**

Operation: Branch on Z = 0

N Z C I D V

(Ref: 4.1.1.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE Oper	D0	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BPL**BPL** Branch on result plus**BPL**

Operation: Branch on N = 0

N Z C I D V

(Ref: 4.1.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL Oper	10	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BRK**BRK** Force Break**BRK**

Operation: Forced Interrupt PC + 2 + P +

N Z C I D V

(Ref: 9.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	00	1	7

1. A BRK command cannot be masked by setting I.

BVC**BVC** *Branch on overflow clear***BVC**

Operation: Branch on V = 0

N Z C I D V

(Ref: 4.1.1.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC Oper	50	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BVS**BVS** *Branch on overflow set***BVS**

Operation: Branch on V = 1

N Z C I D V

(Ref: 4.1.1.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS Oper	70	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

CLC**CLC** *Clear carry flag***CLC**

Operation: 0 → C

N Z C I D V

(Ref: 3.0.2)

-- 0 ---

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLC	18	1	2

CLD**CLD** *Clear decimal mode***CLD**Operation: $\emptyset \rightarrow D$

N Z C I D V

- - - - \emptyset -

(Ref: 3.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI**CLI** *Clear interrupt disable bit***CLI**Operation: $\emptyset \rightarrow I$

N Z C I D V

- - - - \emptyset - -

(Ref: 3.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV**CLV** *Clear overflow flag***CLV**Operation: $\emptyset \rightarrow V$

N Z C I D V

- - - - - \emptyset

(Ref: 3.6.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP**CMP** Compare memory and accumulator**CMP**

Operation: A - M

N Z C I D V

(Ref: 4.2.1)

✓ ✓ / - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CMP #Oper	C9	2	2
Zero Page	CMP Oper	C5	2	3
Zero Page, X	CMP Oper, X	D5	2	4
Absolute	CMP Oper	CD	3	4
Absolute, X	CMP Oper, X	DD	3	4*
Absolute, Y	CMP Oper, Y	D9	3	4*
(Indirect, X)	CMP (Oper, X)	C1	2	6
(Indirect), Y	CMP (Oper), Y	D1	2	5*

* Add 1 if page boundary is crossed.

CPX**CPX** Compare Memory and Index X**CPX**

Operation: X - M

N Z C I D V

(Ref: 7.8)

✓ ✓ / - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPX #Oper	E0	2	2
Zero Page	CPX Oper	E4	2	3
Absolute	CPX Oper	EC	3	4

CPY**CPY** Compare memory and index Y**CPY**

Operation: Y - M

N Z C I D V

(Ref: 7.9)

✓ ✓ / - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPY #Oper	C0	2	2
Zero Page	CPY Oper	C4	2	3
Absolute	CPY Oper	CC	3	4

DEC**DEC** *Decrement memory by one***DEC**Operation: $M - 1 \rightarrow M$

N Z C I D V

✓ / - - - -

(Ref: 10.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC Oper	C6	2	5
Zero Page, X	DEC Oper, X	D6	2	6
Absolute	DEC Oper	CE	3	6
Absolute, X	DEC Oper, X	DE	3	7

DEX**DEX** *Decrement index X by one***DEX**Operation: $X - 1 \rightarrow X$

N Z C I D V

✓ / - - - -

(Ref: 7.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY**DEY** *Decrement index Y by one***DEY**Operation: $Y - 1 \rightarrow Y$

N Z C I D V

✓ / - - - -

(Ref: 7.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR**EOR** "Exclusive-Or" memory with accumulator**EOR**Operation: $A \nabla M \rightarrow A$

N Z C I D V

(Ref: 2.2.3.2)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	EOR #Oper	49	2	2
Zero Page	EOR Oper	45	2	3
Zero Page, X	EOR Oper, X	55	2	4
Absolute	EOR Oper	4D	3	4
Absolute, X	EOR Oper, X	5D	3	4*
Absolute, Y	EOR Oper, Y	59	3	4*
(Indirect, X)	EOR (Oper, X)	41	2	6
(Indirect), Y	EOR (Oper), Y	51	2	5*

* Add 1 if page boundary is crossed.

INC**INC** Increment memory by one**INC**Operation: $M + 1 \rightarrow M$

N Z C I D V

(Ref: 10.6)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	INC Oper	E6	2	5
Zero Page, X	INC Oper, X	F6	2	6
Absolute	INC Oper	EE	3	6
Absolute, X	INC Oper, X	FE	3	7

INX**INX** Increment Index X by one**INX**Operation: $X + 1 \rightarrow X$

N Z C I D V

(Ref: 7.4)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY*INY Increment Index Y by one***INY**Operation: $Y + 1 \rightarrow Y$

N Z C I D V

(Ref: 7.5)

✓ / - - - -

Addressing Mode	Assembly Language Form *	OP CODE	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP*JMP Jump to new location***JMP**Operation: $(PC + 1) \rightarrow PCL$

N Z C I D V

 $(PC + 2) \rightarrow PCH$ (Ref: 4.0.2)
 (Ref: 9.8.1)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP Oper	4C	3	3
Indirect	JMP (Oper)	6C	3	5

JSR*JSR Jump to new location saving return address***JSR**Operation: $PC + 2 \rightarrow$, $(PC + 1) \rightarrow PCL$

N Z C I D V

 $(PC + 2) \rightarrow PCH$
 (Ref: 8.1)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR Oper	20	3	6

LDA**LDA** Load accumulator with memory**LDA**Operation: $M \rightarrow A$

N Z C I D V

✓ / - - - -

(Ref: 2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDA #Oper	A9	2	2
Zero Page	LDA Oper	A5	2	3
Zero Page, X	LDA Oper, X	B5	2	4
Absolute	LDA Oper	AD	3	4
Absolute, X	LDA Oper, X	BD	3	4*
Absolute, Y	LDA Oper, Y	B9	3	4*
(Indirect, X)	LDA (Oper, X)	A1	2	6
(Indirect), Y	LDA (Oper), Y	B1	2	5*

* Add 1 if page boundary is crossed.

LDX**LDX** Load index X with memory**LDX**Operation: $M \rightarrow X$

N Z C I D V

✓ / - - - -

(Ref: 7.0)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDX # Oper	A2	2	2
Zero Page	LDX Oper	A6	2	3
Zero Page, Y	LDX Oper, Y	B6	2	4
Absolute	LDX Oper	AE	3	4
Absolute, Y	LDX Oper, Y	BE	3	4*

* Add 1 when page boundary is crossed.

LDY**LDY** Load index Y with memory**LDY**

Operation: M → Y

N Z C I D V

✓ / - - - -

(Ref: 7.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDY #Oper	A0	2	2
Zero Page	LDY Oper	A4	2	3
Zero Page, X	LDY Oper, X	B4	2	4
Absolute	LDY Oper	AC	3	4
Absolute, X	LDY Oper, X	BC	3	4*

* Add 1 when page boundary is crossed.

LSR**LSR** Shift right one bit (memory or accumulator)**LSR**Operation: 0 →

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

 → C

N Z C I D V

0 ✓ / - - - -

(Ref: 10.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	LSR A	4A	1	2
Zero Page	LSR Oper	46	2	5
Zero Page, X	LSR Oper, X	56	2	6
Absolute	LSR Oper	4E	3	6
Absolute, X	LSR Oper, X	5E	3	7

NOP**NOP** No operation**NOP**

Operation: No Operation (2 cycles)

N Z C I D V

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA**ORA** "OR" memory with accumulator**ORA**

Operation: A V M → A

N Z C I D V

(Ref: 2.2.3.1)

✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ORA #Oper	09	2	2
Zero Page	ORA Oper	05	2	3
Zero Page, X	ORA Oper, X	15	2	4
Absolute	ORA Oper	0D	3	4
Absolute, X	ORA Oper, X	1D	3	4*
Absolute, Y	ORA Oper, Y	19	3	4*
(Indirect, X)	ORA (Oper, X)	01	2	6
(Indirect), Y	ORA (Oper), Y	11	2	5

* Add 1 on page crossing

PHA**PHA** Push accumulator on stack**PHA**

Operation: A →

N Z C I D V

(Ref: 8.5)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP**PHP** Push processor status on stack**PHP**

Operation: P →

N Z C I D V

(Ref: 8.11)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	08	1	3

PLA**PLA** Pull accumulator from stack**PLA**

Operation: A ↑

N Z C I D V

(Ref: 8.6)

✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP**PLP** Pull processor status from stack**PLP**

Operation: P ↑

N Z C I D V

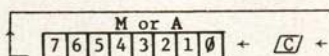
(Ref: 8.12)

From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL**ROL** Rotate one bit left (memory or accumulator)**ROL**

Operation:

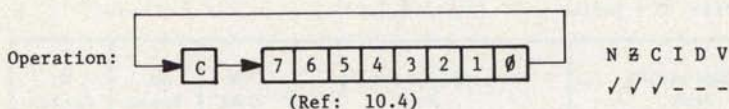


N Z C I D V

✓ ✓ ✓ - - -

(Ref: 10.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROL A	2A	1	2
Zero Page	ROL Oper	26	2	5
Zero Page, X	ROL Oper, X	36	2	6
Absolute	ROL Oper	2E	3	6
Absolute, X	ROL Oper, X	3E	3	7

ROR**ROR** Rotate one bit right (memory or accumulator)**ROR**

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR Oper	66	2	5
Zero Page,X	ROR Oper,X	76	2	6
Absolute	ROR Oper	6E	3	6
Absolute,X	ROR Oper,X	7E	3	7

Note: ROR instruction will be available on MCS650X micro-processors after June, 1976.

RTI**RTI** Return from interrupt**RTI**

Operation: P ← PC+

N Z C I D V

(Ref: 9.6)

From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	40	1	6

RTS**RTS** Return from subroutine**RTS**

Operation: PC+, PC + 1 → PC

N Z C I D V

(Ref: 8.2)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	60	1	6

SBC**SBC** Subtract memory from accumulator with borrow**SBC**Operation: $A - M - \bar{C} + A$

N Z C I D V

Note: \bar{C} = Borrow

(Ref: 2.2.2)

✓ ✓ ✓ - - ✓

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	SBC #Oper	E9	2	2
Zero Page	SBC Oper	E5	2	3
Zero Page, X	SBC Oper, X	F5	2	4
Absolute	SBC Oper	ED	3	4
Absolute, X	SBC Oper, X	FD	3	4*
Absolute, Y	SBC Oper, Y	F9	3	4*
(Indirect, X)	SBC (Oper, X)	E1	2	6
(Indirect), Y	SBC (Oper), Y	F1	2	5*

* Add 1 when page boundary is crossed.

SEC**SEC** Set carry flag**SEC**Operation: $1 \rightarrow C$

N Z C I D V

(Ref: 3.0.1)

- - 1 - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED**SED** Set decimal mode**SED**Operation: $1 \rightarrow D$

N Z C I D V

(Ref: 3.3.1)

- - - - 1 -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI**SEI** *Set interrupt disable status***SEI**Operation: $1 \rightarrow I$

N Z C I D V

--- 1 ---

(Ref: 3.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA**STA** *Store accumulator in memory***STA**Operation: $A \rightarrow M$

N Z C I D V

(Ref: 2.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA Oper	85	2	3
Zero Page, X	STA Oper, X	95	2	4
Absolute	STA Oper	8D	3	4
Absolute, X	STA Oper, X	9D	3	5
Absolute, Y	STA Oper, Y	99	3	5
(Indirect, X)	STA (Oper, X)	81	2	6
(Indirect), Y	STA (Oper), Y	91	2	6

STX**STX** *Store index X in memory***STX**Operation: $X \rightarrow M$

N Z C I D V

(Ref: 7.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX Oper	86	2	3
Zero Page, Y	STX Oper, Y	96	2	4
Absolute	STX Oper	8E	3	4

STY**STY** Store index Y in memory**STY**

Operation: Y → M

N Z C I D V

(Ref: 7.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY Oper	84	2	3
Zero Page, X	STY Oper, X	94	2	4
Absolute	STY Oper	8C	3	4

TAX**TAX** Transfer accumulator to index X**TAX**

Operation: A → X

N Z C I D V

✓ / -----

(Ref: 7.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY**TAY** Transfer accumulator to index Y**TAY**

Operation: A → Y

N Z C I D V

✓ / -----

(Ref: 7.13)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX*TSX Transfer stack pointer to index X***TSX**Operation: $S \rightarrow X$

N Z C I D V

(Ref: 8.9)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA*TXA Transfer index X to accumulator***TXA**Operation: $X \rightarrow A$

N Z C I D V

(Ref: 7.12)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS*TXS Transfer index X to stack pointer***TXS**Operation: $X \rightarrow S$

N Z C I D V

(Ref: 8.8)

- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA*TYA Transfer index Y to accumulator***TYA**Operation: $Y \rightarrow A$

N Z C I D V

(Ref: 7.14)

✓ / - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

INSTRUCTION ADDRESSING MODES AND

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7
BCC	2**	.	.	.
BCS	2**	.	.	.
BEQ	2**	.	.	.
BIT	.	.	3	.	.	4
BMI	2**	.	.	.
BNE	2**	.	.	.
BPL	2**	.	.	.
BRK
BVC	2**	.	.	.
BVS	2**	.	.	.
CLC	2
CLD	2
CLI	2
CLV	2
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4
CPY	.	2	3	.	.	4
DEC	.	.	5	6	.	6	7
DEX	2
DEY	2
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5	.
INC	.	.	5	6	.	6	7
INX	2
INY	2
JMP	3	5

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional

RELATED EXECUTION TIMES (in clock cycles)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
JSR	6
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*
LDY	.	2	3	4	.	4	4*
LSR	2	.	5	6	.	6	7
NOP	2
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	3
PHP	3
PLA	4
PLP	4
ROL	2	.	5	6	.	6	7
ROR	2	.	5	6	.	6	7
RTI	6
RTS	6
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	2
SED	2
SEI	2
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX*	.	.	3	.	4	4
STY**	.	.	3	4	.	4
TAX	2
TAY	2
TSX	2
TXA	2
TXS	2
TYA	2

if branching operation crosses page boundary

00 - BRK	20 - JSR
01 - ORA - (Indirect,X)	21 - AND - (Indirect,X)
02 - Future Expansion	22 - Future Expansion
03 - Future Expansion	23 - Future Expansion
04 - Future Expansion	24 - BIT - Zero Page
05 - ORA - Zero Page	25 - AND - Zero Page
06 - ASL - Zero Page	26 - ROL - Zero Page
07 - Future Expansion	27 - Future Expansion
08 - PHP	28 - PLP
09 - ORA - Immediate	29 - AND - Immediate
0A - ASL - Accumulator	2A - ROL - Accumulator
0B - Future Expansion	2B - Future Expansion
0C - Future Expansion	2C - BIT - Absolute
0D - ORA - Absolute	2D - AND - Absolute
0E - ASL - Absolute	2E - ROL - Absolute
0F - Future Expansion	2F - Future Expansion
10 - BPL	30 - BMI
11 - ORA - (Indirect),Y	31 - AND - (Indirect),Y
12 - Future Expansion	32 - Future Expansion
13 - Future Expansion	33 - Future Expansion
14 - Future Expansion	34 - Future Expansion
15 - ORA - Zero Page,X	35 - AND - Zero Page,X
16 - ASL - Zero Page,X	36 - ROL - Zero Page,X
17 - Future Expansion	37 - Future Expansion
18 - CLC	38 - SEC
19 - ORA - Absolute,Y	39 - AND - Absolute,Y
1A - Future Expansion	3A - Future Expansion
1B - Future Expansion	3B - Future Expansion
1C - Future Expansion	3C - Future Expansion
1D - ORA - Absolute,X	3D - AND - Absolute,X
1E - ASL - Absolute,X	3E - ROL - Absolute,X
1F - Future Expansion	3F - Future Expansion

40 - RTI	60 - RTS
41 - EOR - (Indirect,X)	61 - ADC - (Indirect,X)
42 - Future Expansion	62 - Future Expansion
43 - Future Expansion	63 - Future Expansion
44 - Future Expansion	64 - Future Expansion
45 - EOR - Zero Page	65 - ADC - Zero Page
46 - LSR - Zero Page	66 - ROR - Zero Page
47 - Future Expansion	67 - Future Expansion
48 - PHA	68 - PLA
49 - EOR - Immediate	69 - ADC - Immediate
4A - LSR - Accumulator	6A - ROR - Accumulator
4B - Future Expansion	6B - Future Expansion
4C - JMP - Absolute	6C - JMP - Indirect
4D - EOR - Absolute	6D - ADC - Absolute
4E - LSR - Absolute	6E - ROR - Absolute
4F - Future Expansion	6F - Future Expansion
50 - BVC	70 - BVS
51 - EOR - (Indirect),Y	71 - ADC - (Indirect),Y
52 - Future Expansion	72 - Future Expansion
53 - Future Expansion	73 - Future Expansion
54 - Future Expansion	74 - Future Expansion
55 - EOR - Zero Page,X	75 - ADC - Zero Page,X
56 - LSR - Zero Page,X	76 - ROR - Zero Page,X
57 - Future Expansion	77 - Future Expansion
58 - CLI	78 - SEI
59 - EOR - Absolute,Y	79 - ADC - Absolute,Y
5A - Future Expansion	7A - Future Expansion
5B - Future Expansion	7B - Future Expansion
5C - Future Expansion	7C - Future Expansion
5D - EOR - Absolute,X	7D - ADC - Absolute,X
5E - LSR - Absolute,X	7E - ROR - Absolute,X
5F - Future Expansion	7F - Future Expansion

80 - Future Expansion
 81 - STA - (Indirect,X)
 82 - Future Expansion
 83 - Future Expansion
 84 - STY - Zero Page
 85 - STA - Zero Page
 86 - STX - Zero Page
 87 - Future Expansion
 88 - DEY
 89 - Future Expansion
 8A - TXA
 8B - Future Expansion
 8C - STY - Absolute
 8D - STA - Absolute
 8E - STX - Absolute
 8F - Future Expansion
 90 - BCC
 91 - STA - (Indirect),Y
 92 - Future Expansion
 93 - Future Expansion
 94 - STY - Zero Page,X
 95 - STA - Zero Page,X
 96 - STX - Zero Page,Y
 97 - Future Expansion
 98 - TYA
 99 - STA - Absolute,Y
 9A - TXS
 9B - Future Expansion
 9C - Future Expansion
 9D - STA - Absolute,X
 9E - Future Expansion
 9F - Future Expansion

A0 - LDY - Immediate
 A1 - LDA - (Indirect,X)
 A2 - LDX - Immediate
 A3 - Future Expansion
 A4 - LDY - Zero Page
 A5 - LDA - Zero Page
 A6 - LDX - Zero Page
 A7 - Future Expansion
 A8 - TAY
 A9 - LDA - Immediate
 AA - TAX
 AB - Future Expansion
 AC - LDY - Absolute
 AD - LDA - Absolute
 AE - LDX - Absolute
 AF - Future Expansion
 B0 - BCS
 B1 - LDA - (Indirect),Y
 B2 - Future Expansion
 B3 - Future Expansion
 B4 - LDY - Zero Page,X
 B5 - LDA - Zero Page,X
 B6 - LDX - Zero Page,Y
 B7 - Future Expansion
 B8 - CLV
 B9 - LDA - Absolute,Y
 BA - TSX
 BB - Future Expansion
 BC - LDY - Absolute,X
 BD - LDA - Absolute,X
 BE - LDX - Absolute,Y
 BF - Future Expansion

C0 - CPY - Immediate
 C1 - CMP - (Indirect,X)
 C2 - Future Expansion
 C3 - Future Expansion
 C4 - CPY - Zero Page
 C5 - CMP - Zero Page
 C6 - DEC - Zero Page
 C7 - Future Expansion
 C8 - INY
 C9 - CMP - Immediate
 CA - DEX
 CB - Future Expansion
 CC - CPY - Absolute
 CD - CMP - Absolute
 CE - DEC - Absolute
 CF - Future Expansion
 D0 - BNE
 D1 - CMP - (Indirect),Y
 D2 - Future Expansion
 D3 - Future Expansion
 D4 - Future Expansion
 D5 - CMP - Zero Page,X
 D6 - DEC - Zero Page,X
 D7 - Future Expansion
 D8 - CLD
 D9 - CMP - Absolute,Y
 DA - Future Expansion
 DB - Future Expansion
 DC - Future Expansion
 DD - CMP - Absolute,X
 DE - DEC - Absolute,X
 DF - Future Expansion

E0 - CPX - Immediate
 E1 - SBC - (Indirect,X)
 E2 - Future Expansion
 E3 - Future Expansion
 E4 - CPX - Zero Page
 E5 - SBC - Zero Page
 E6 - INC - Zero Page
 E7 - Future Expansion
 E8 - INX
 E9 - SBC - Immediate
 EA - NOP
 EB - Future Expansion
 EC - CPX - Absolute
 ED - SBC - Absolute
 EE - INC - Absolute
 EF - Future Expansion
 F0 - BEQ
 F1 - SBC - (Indirect),Y
 F2 - Future Expansion
 F3 - Future Expansion
 F4 - Future Expansion
 F5 - SBC - Zero Page,X
 F6 - INC - Zero Page,X
 F7 - Future Expansion
 F8 - SED
 F9 - SBC - Absolute,Y
 FA - Future Expansion
 FB - Future Expansion
 FC - Future Expansion
 FD - SBC - Absolute,X
 FE - INC - Absolute,X
 FF - Future Expansion

SPECIAL TIPS FOR BEGINNERS

Learning to write machine language programs is a discipline which is very useful in programming. Since machine language is at the same level as the internal workings of the machine, your brain is stretched that much further, when trying to organize things in your mind and in the VIC-20.

One of the best ways to learn machine language is to look at other people's machine language programs. These are published all the time, in magazines and newsletters, even if the article is for a different computer that also has the 6502 microprocessor (there are many). You should make sure that you thoroughly understand the code that you look at. This may require perseverance, especially when you see a new technique that you have never come across before. This can be infuriating, but if patience prevails, you will be the VICtor (sorry about that).

Having looked at other machine language programs, you **MUST** write your own. These may be utilities for your BASIC programs, or may be an all machine language program. You should also use the utilities that are available, either in your computer, or in a program, that aid you in writing, editing, or tracking down errors in a machine language program. An example would be the KERNAL, which allows you to check the keyboard, print text, control peripheral devices like disk drives, printers, modems, etc., manage memory and the screen. It is extremely powerful and it is advised strongly that it is used (refer to KERNAL section).

ADVANTAGES OF WRITING PROGRAMS IN MACHINE LANGUAGE

1. Speed—Machine language is hundreds, and in some cases thousands, of times faster than a high level language such as BASIC.

2. Tightness—A machine language program can be made totally "watertight," i.e., the user can be made to do **ONLY** what the program allows, and no more. With a high level language, you are relying on the user not "crashing" the BASIC interpreter by entering, for example, a zero which later causes a:

```
?DIVISION BY ZERO
ERROR IN LINE 830
READY.
```

*

In essence, the computer belongs to the machine language programmer.

APPROACHING A LARGE TASK

When approaching a large task in machine language, a certain amount of subconscious thought has usually taken place, in thinking about how certain processes would be implemented in machine language. When the task is started, it is usually a good idea to set out on paper block diagrams of memory usage, functional modules of code required, and a program flow. Let's say that we wanted to write a roulette game in machine language. We can outline this as shown below.

```
Display title
Ask if player requires instructions
YES—display them—Go to START
NO—Go to START
START Initialize everything
MAIN display roulette table
Take in bets
Spin wheel
Slow wheel to stop
Check bets with result
Inform player
Player any money left
YES—Go to MAIN
NO—Inform user!, and Go to START
```

This is the main outline, which, as each module is approached, can then be broken down further. If you look upon a large indigestible problem as something that once broken down into small enough pieces can all be eaten, then this will enable you to approach something that seems impossible, and you will be surprised at how swiftly it all falls into place. This process obviously improves with practice, but KEEP TRYING.

MEMORY MAPS

The following memory maps provide a guide which shows which special locations are set aside for use by the VIC's operating system . . . and what those locations are used for.

Memory Map

HEX	DECIMAL	DESCRIPTION
0000	0	Jump for USR
0001-0002	1-2	Vector for USR
0003-0004	3-4	Float-Fixed vector
0005-0006	5-6	Fixed-Float vector
0007	7	Search character
0008	8	Scan-quotes flag
0009	9	TAB column save
000A	10	0 = LOAD, 1 = VERIFY
000B	11	Input buffer pointer/# subscript
000C	12	Default DIM flag
000D	13	Type: FF = string, 00 = numeric
000E	14	Type: 80 = integer, 00 = floating point
000F	15	DATA scan/LIST quote/memory flag
0010	16	Subscript/FNx flag
0011	17	0 = INPUT; \$40 = GET; \$98 = READ
0012	18	ATN sign/Comparison eval flag
0013	19	Current I/O prompt flag
*0014-0015	20-21	Integer value
0016	22	Pointer: temporary string stack
0017-0018	23-24	Last temp string vector
0019-0021	25-33	Stack for temporary strings
0022-0025	34-37	Utility pointer area
0026-002A	38-42	Product area for multiplication
*002B-002C	43-44	Pointer: Start of Basic
*002D-002E	45-46	Pointer: Start of Variables
*002F-0030	47-48	Pointer: Start of Arrays
*0031-0032	49-50	Pointer: End of Arrays
*0033-0034	51-52	Pointer: String storage (moving down)
0035-0036	53-54	Utility string pointer
*0037-0038	55-56	Pointer: Limit of memory
0039-003A	57-58	Current Basic line number
003B-003C	59-60	Previous Basic line number
003D-003E	61-62	Pointer: Basic statement for CONT
003F-0040	63-64	Current DATA line number
0041-0042	65-66	Current DATA address
*0043-0044	67-68	Input vector

* Useful memory location

HEX	DECIMAL	DESCRIPTION
0045-0045	69-70	Current variable name
0047-0048	71-72	Current variable address
0049-004A	73-74	Variable pointer for FOR/NEXT
004B-004C	75-76	Y-save; op-save; Basic pointer save
004D	77	Comparison symbol accumulator
004E-0053	78-83	Misc work area, pointers, etc
0054-0056	84-86	Jump vector for functions
0057-0060	87-96	Misc numeric work area
*0061	97	Accum#1: Exponent
*0062-0065	98-101	Accum#1: Mantissa
*0066	102	Accum#1: Sign
0067	103	Series evaluation constant pointer
0068	104	Accum#1 hi-order (overflow)
*0069-006E	105-110	Accum#2: Exponent, etc.
006F	111	Sign comparison, Acc#1 vs #2
0070	112	Accum#1 lo-order (rounding)
0071-0072	113-114	Cassette buffer length/Series pointer
*0073-008A	115-138	CHRGET subroutine (get BASIC char)
007A-007B	122-123	Basic pointer (within subroutine)
008B-008F	139-143	RND seed value
*0090	144	Status word ST
0091	145	Keyswitch PIA: STOP and RVS flags
0092	146	Timing constant for tape
0093	147	Load = 0, Verify = 1
0094	148	Serial output: deferred char flag
0095	149	Serial deferred character
0096	150	Tape EOT received
0097	151	Register save
*0098	152	How many open files
*0099	153	Input device (normally 0)
*009A	154	Output (CMD) device, normally 3
009B	155	Tape character parity
009C	156	Byte-received flag
009D	157	Direct = \$80/RUN = 0 output control
009E	158	Tape Pass 1 error log/char buffer
009F	159	Tape Pass 2 error log corrected
*00A0-00A2	160-162	Jiffy Clock (HML)
00A3	163	Serial bit count/EOI flag
00A4	164	Cycle count
00A5	165	Countdown, tape write/bit count
00A6	166	Pointer: tape buffer
00A7	167	Tape Write ldr count/Read pass/inbit
00A8	168	Tape Write new byte/Read error/inbit cnt
00A9	169	Write start bit/Read bit err/stbit

* Useful memory location

HEX	DECIMAL	DESCRIPTION
00AA	170	Tape Scan;Cnt;Ld;End/byte assy
00AB	171	Write lead length/Rd checksum/parity
00AC-00AD	172-173	Pointer: tape buffer, scrolling
00AE-00AF	174-175	Tape end addresses/End of program
00B0-00B1	176-177	Tape timing constants
*00B2-00B3	178-179	Pointer: start of tape buffer
00B4	180	Tape timer (1=enable); bit cnt
00B5	181	Tape EOT/RS-232 next bit to send
00B6	182	Read character error/outbyte buffer
*00B7	183	# characters in file name
*00B8	184	Current logical file
*00B9	185	Current secondary address
*00BA	186	Current device
*00BB-00BC	187-188	Pointer: to file name
00BD	189	Write shift word/Read input char
00BE	190	# blocks remaining to Write/Read
00BF	191	Serial word buffer
00C0	192	Tape motor interlock
00C1-00C2	193-194	I/O start addresses
00C3-00C4	195-196	KERNAL setup pointer
*00C5	197	Current key pressed
*00C6	198	# chars in keyboard buffer
*00C7	199	Screen reverse flag
00C8	200	Pointer: End-of-line for input
00C9-00CA	201-202	Input cursor log (row, column)
*00CB	203	Which key: 64 if no key
00CC	204	cursor enable (0=flash cursor)
00CD	205	Cursor timing countdown
00CE	206	Character under cursor
00CF	207	Cursor in blink phase
00D0	208	Input from screen/from keyboard
*00D1-00D2	209-210	Pointer to screen line
*00D3	211	Position of cursor on above line
00D4	212	0=direct cursor, else programmed
*00D5	213	Current screen line length
*00D6	214	Row where cursor lives
00D7	215	Last inkey/checksum/buffer
*00D8	216	# of INSERTs outstanding
*00D9-00F0	217-240	Screen line link table
00F1	241	Dummy screen link
00F2	242	Screen row marker
*00F3-00F4	243-244	Screen color pointer
00F5-00F6	245-246	Keyboard pointer
00F7-00F8	247-248	RS-232 Rcv pointer
00F9-00FA	249-250	RS-232 Tx pointer

* Useful memory location

HEX	DECIMAL	DESCRIPTION
*00FB-00FE	251-254	Operating system free zero page space
00FF	255	Basic storage
0100-010A	256-266	Floating to ASCII work area
0100-013E	256-318	Tape error log
0100-01FF	256-511	Processor stack area
*0200-0258	512-600	Basic input buffer
*0259-0262	601-610	Logical file table
*0263-026C	611-620	Device # table
*026D-0276	621-630	Secondary Address table
*0277-0280	631-640	Keyboard buffer
*0281-0282	641-642	Start of memory for op system
*0283-0284	643-644	Top of memory for op system
0285	645	Serial bus timeout flag
*0286	646	Current color code
0287	647	Color under cursor
*0288	648	Screen memory page
*0289	649	Max size of keyboard buffer
*028A	650	Key repeat (128=repeat all keys)
*028B	651	Repeat speed counter
028C	652	Repeat delay counter
*028D	653	Keyboard Shift/Control flag
028E	654	Last keyboard shift pattern
028F-0290	655-656	Pointer: decode logic
*0291	657	Shift mode switch (0=enabled, 128-locked)
0292	658	Auto scroll down flag (0=on, <>0=off)
0293	659	RS-232 control register
0294	660	RS-232 command register
0295-0296	661-662	Nonstandard (Bit time/2-100)
0297	663	RS-232 status register
0298	664	Number of bits to send
0299-029A	665-666	Baud rate (full) bit time
029B	667	RS-232 receive pointer
029C	668	RS-232 input pointer
029D	669	RS-232 transmit pointer
029E	670	RS-232 output pointer
029F-02A0	671-672	Holds IRQ during tape operations
02A1-02FF	673-767	Program indirects
*0300-0301	768-769	Error message link
0302-0303	770-771	Basic warm start link
0304-0305	772-773	Crunch Basic tokens link
0306-0307	774-775	Print tokens link
0308-0309	776-777	Start new Basic code link

* Useful memory location

HEX	DECIMAL	DESCRIPTION
030A-030B	778-779	Get arithmetic element link
030C	780	Storage for 6502 .A register
030D	781	Storage for 6502 .X register
030E	782	Storage for 6502 .Y register
030F	783	Storage for 6502 .P register
0310-0313	784-787	??
0314-0315	788-789	Hardware (IRQ) interrupt vector (EABF)
0316-0317	790-791	Break interrupt vector (FED2)
0318-0319	792-793	NMI interrupt vector (FEAD)
031A-031B	794-795	OPEN vector (F40A)
031C-031D	796-797	CLOSE vector (F34A)
031E-031F	798-799	Set-input vector (F2C7)
0320-0321	800-801	Set-output vector (F309)
0322-0323	802-803	Restore I/O vector (F3F3)
0324-0325	804-805	INPUT vector (F20E)
0326-0327	806-807	Output vector (F27A)
0328-0329	808-809	Test-STOP vector (F770)
032A-032B	810-811	GET vector (F1F5)
032C-032D	812-813	Abort I/O vector (F3EF)
032E-032F	814-815	user vector (FED2)
0330-0331	816-817	Link to load RAM (F549)
0332-0333	818-819	Link to save RAM (F685)
0334-033B	820-827	??
*033C-03FB	828-1019	Cassette buffer
0400-0FFF	1024-4095	3K expansion RAM area
1000-1DFF	4096-7679	User Basic area
1E00-1FFF	7680-8191	Screen memory
2000-3FFF	8192-16383	8K expansion RAM/ROM block 1
4000-5FFF	16384-24575	8K expansion RAM/ROM block 2
6000-7FFF	24576-32767	8K expansion RAM/ROM block 3

NOTE: When additional memory is added to block 1 (and 2 and 3), the KERNAL relocates the following things for BASIC:

1000-11FF	4096-4607	Screen memory
1200-?	4608-?	User Basic area
9400-95FF	37888 = 38399	Color RAM

8000-8FFF	32768-36863	4K Character generator ROM
8000-83FF	32768-33791	Upper case and graphics
8400-87FF	33792-33815	Reversed upper case and graphics
8800-8BFF	33816-35839	Upper and lower case
8C00-8FFF	35840-36863	Reversed upper and lower case
9000-93FF	36864-37887	I/O BLOCK O

* Useful memory location

HEX	DECIMAL	DESCRIPTION
9000-900F	36864-36879	Address of VIC chip registers
9000	36864	bits 0-6 horizontal centering bit 7 sets interlace scan
9001	36865	vertical centering
9002	36866	bits 0-6 set # of columns bit 7 is part of video matrix address
9003	36867	bits 1-6 set # of rows bit 0 sets 8×8 or 16×8 chars
9004	36868	TV raster beam line
9005	36869	bits 0-3 start of character memory (default=0) bits 4-7 is rest of video address (default=F) BITS 3,2,1,0 CM starting address

		HEX	DEC
		0000	32768
	ROM	0001	33792
		0010	34816
		0011	35840
	RAM	1000	0000
		1001	xxxx
		1010	xxxx
		1011	xxxx
		1100	4096
		1101	5120
		1110	6144
		1111	7168
9006	36870	horizontal position of light pen	
9007	36871	vertical position of light pen	
9008	36872	Digitized value of paddle X	
9009	36873	Digitized value of paddle Y	
900A	36874	Frequency for oscillator 1 (low) (on: 128-255)	
900B	36875	Frequency for oscillator 2 (medium) (on: 128-255)	
900C	36876	Frequency for oscillator 3 (high) on: 128-255)	
900D	36877	Frequency of noise source	
900E	36878	bit 0-3 sets volume of all sound bits 4-7 are auxiliary color information	
900F	36879	Screen and border color register bits 4-7 select background color bits 0-2 select border color bit 3 selects inverted or normal mode	

HEX	DECIMAL	DESCRIPTION
9110-911F	37136-37151	6522 VIA#1
9110	37136	Port B output register (user port and RS-232 lines)
	PIN ID	6522 DESCRIPTION EIA ABV ID
	C	PB0 Received data (BB) Sin
	D	PB1 Request to Send (CA) RTS
	E	PB2 Data terminal ready (CD) DTR
	F	PB3 Ring indicator (CE) RI
	H	PB4 Received line signal (CF) DCD
	J	PB5 Unassigned () XXX
	K	PB6 Clear to send (CB) CTS
	L	PB7 Data set ready (CC) DSR
	B	CB1 Interrupt for Sin (BB) Sin
	M	CB2 Transmitted data (BA) Sout
	A	GND Protective ground (AA) GND
	N	GND Signal ground (AB) GND
9111	37137	Port A output register (PA0) Bit 0 = Serial CLK IN (PA1) Bit 1 = Serial DATA IN (PA2) Bit 2 = Joy 0 (PA3) Bit 3 = Joy 1 (PA4) Bit 4 = Joy 2 (PA5) Bit 5 = Lightpen/Fire button (PA6) Bit 6 = Cassette switch sense (PA7) Bit 7 = Serial ATN out
9112	37138	Data direction register B
9113	37139	Data direction register A
9114	37140	Timer 1 low byte
9115	37141	Timer 1 high byte & counter
9116	37142	Timer 1 low byte
9117	37143	Timer 1 high byte
9118	37144	Timer 2 low byte
9119	37145	Timer 2 high byte
911A	37146	Shift register
911B	37147	Auxiliary control register
911C	37148	Peripheral control register (CA1, CA2, CB1, CB2) CA1 = restore key (Bit 0) CA2 = cassette motor control (Bits 1-3) CB1 = interrupt signal for received RS-232 data (Bit 4) CB2 = transmitted RS-232 data (Bits 5-7)
911D	37149	Interrupt flag register



HEX	DECIMAL	DESCRIPTION
911E	37150	Interrupt enable register
911F	37151	Port A (Sense cassette switch)
9120-912F	37152-37167	6522 VIA#2
9120	37152	Port B output register keyboard column scan (PB3) Bit 3 = cassette write line (PB7) Bit 7 = Joy 3
9121	37153	Port A output register keyboard row scan
9122	37154	Data direction register B
9123	37155	Data direction register A
9124	37156	Timer 1, low byte latch
9125	37157	Timer 1, high byte latch
9126	37158	Timer 1, low byte counter
9127	37159	Timer 1, high byte counter timer 1 is used for the 60 time/second interrupt
9128	37160	Timer 2, low byte latch
9129	37161	Timer 2, high byte latch
912A	37162	Shift register
912B	37163	Auxiliary control register
912C	37164	Peripheral control register CA1 Cassette read line (Bit 0) CA2 Serial clock out (Bits 1-3) CB1 Serial SRQ IN (Bit 4) CB2 Serial data out (Bits 5-7)
912D	37165	Interrupt flag register
912E	37166	Interrupt enable register
912F	37167	Port A output register
9400-95FF	37888-38399	location of COLOR RAM with additional RAM at blk 1
9600-97FF	38400-38911	Normal location of COLOR RAM
9800-9BFF	38912-39935	I/O block 2
9C00-9FFF	39936-40959	I/O block 3
A000-BFFF	40960-49152	8K decoded block for expansion ROM
C000-DFFF	49152-57343	8K Basic ROM
E000-FFFF	57344-65535	8K KERNAL ROM

USEFUL MEMORY LOCATIONS

This is a more in-depth guide to some of the memory locations you can use.

HEX	DECIMAL	DESCRIPTION
0014-0015	20-21	Where BASIC stores integer variables used in calculations. The fixed-float and float-fixed routines (vectors at 3-4 and 5-6) use the value in this area.
002B-002C	43-44	The start of the BASIC program in memory. Location 43 contains the low byte, and location 44 has the high byte. To compute the start of BASIC in decimal, use the formula: $\text{PEEK}(43) + 256 * \text{PEEK}(44)$
002D-002E	45-46	The start of the numeric variables, which is usually immediately after the end of the BASIC program.
002F-0030	47-48	The start of arrays in memory, usually immediately following the numeric variables.
0031-0032	49-50	The end of the arrays in memory.
0033-0034	51-52	Bottom of string storage, moving from the top of available memory down to the top of arrays.
0037-0038	55-56	The top of free RAM. By lowering this value, some RAM can be "protected" against BASIC putting values here.
0043-0044	67-68	Jump vector for INPUT statement.
0061-0066	97-102	Floating point accumulator #1 for calculations.
0069-006E	105-110	Floating point accumulator #2.
0073-008A	115-138	The CHRGET subroutine resides here. This routine gets the next BASIC character from machine language.
0090	144	Status word ST.
0098	152	Number of open files.
0099	153	Device number for input, normally 0 (keyboard).
009A	154	Output (CMD) device, normally 3 (screen).

HEX	DECIMAL	DESCRIPTION
00A0-00A2	160-162	3 byte jiffy clock. The TI and TI\$ variables are translations of these locations.
00B2-00B3	178-179	Points to the start of the tape buffer. Can be used as an indirect zero-page jump to a routine in the buffer.
00B7	183	Number of characters in filename.
00B9	185	Which secondary address is currently being used.
00BA	186	Current device number being accessed.
00BB-00BC	187-188	Points to location of filename in memory.
00C5	197	Current key being held down. There will be a 64 here if nothing is held down. If more than 1 key is down, the key with the highest number on the chart is what shows up here.

#	key	#	key	#	key	#	key
0	1	16	none	32	space	48	Q
1	3	17	A	33	Z	49	E
2	5	18	D	34	C	50	T
3	7	19	G	35	B	51	U
4	9	20	J	36	M	52	O
5	+	21	L	37	•	53	@
6	£	22	:	38	none	54	↑
7	DEL	23		39	fl	55	f5
8	←	24	STOP	40	none	56	2
9	W	25	none	41	S	57	4
10	R	26	X	42	F	58	6
11	Y	27	V	43	H	58	8
12	I	28	N	44	K	60	0
13	P	29	,	45	:	61	-
14	*	30	/	46	=	62	HOME
15	RETURN	31		47	f3	63	f7

00C6	198	Number of characters currently in keyboard buffer.
00C7	199	Flag for reverse on/off. A 1 here is on, a 0 is off.
00CB	203	Same as 197.
00D1-00D2	209-210	Address of start of line where cursor is.
00D3	211	Position of cursor on line.
00D5	213	Current screen line length—either 21, 43, 65, or 87.

HEX	DECIMAL	DESCRIPTION
00D6	214	Screen row where cursor is. To change the cursor position, locations 201, 210, 211, and 214 must be changed.
00D8	216	Number of spaces left in INSERT mode. POKEing this to a zero will turn off insert mode.
00D9-00F0	217-240	Screen line link table. A 158 means that the line is finished at the end of that line, and a 30 means that the line continues on the next line.
00F3-00F4	243-244	Pointer to the current space in color memory.
00FB-00FE	251-254	Available locations in zero page.
0200-0258	512-600	BASIC input buffer—where the characters being INPUT will go.
0259-0262	601-610	Logic 1 file table for OPEN files.
0263-026C	611-620	Device # table for OPEN files.
026D-0276	621-630	Secondary address table
0277-0280	631-640	Keyboard buffer. If characters are POKEd in here and location 198 (# of characters in buffer) is changed, it will be as if the characters were typed from the keyboard.
0281-0282	641-642	Start of memory pointer.
0283-0284	643-644	Top of memory pointer.
0286	646	Current color code. This holds the color number that goes into color memory during PRINT operations.
0288	648	Screen memory page. If you want the operating system to know where screen memory is, this must be changed as well as the VIC chip.
0289	649	Maximum size of keyboard buffer. If this is set greater than 10, vital pointers will be destroyed.
028A	650	Keyboard repeat flag. If this is a 0, only cursor controls repeat; if 128, all keys repeat.
028B	651	This determines how long the VIC waits before repeating key.
028D	653	Keyboard SHIFT, CTRL, Commodore flag. The SHIFT sets the 1 bit, Commodore sets the 2 bit, and the CTRL sets the 4 bit.

HEX	DECIMAL	DESCRIPTION
0291	657	Setting this location to 128 will disable switching case, and a 0 here enables the ability to switch.
0300-0301	768-769	This is the jump vector for errors. By changing this vector, a routine can intercept any error condition.
033C-03FB	828-1019	Cassette buffer. This is where data files are held before they are INPUT. When not using files, this is available for POKEing or machine language programs.

THE KERNAL

One of the toughest problems facing programmers in the microcomputer field is the question of what to do when changes are made to the operating system of the computer by the company. Machine language programs which took much time to develop might no longer work, forcing major revisions in the program. To alleviate this problem, Commodore has developed a method of protecting software writers called the **KERNAL**.

Essentially, the KERNAL is a standardized JUMP TABLE to the input, output, and memory management routines in the operating system. The locations of each routine in ROM might change as the system is upgraded. But the KERNAL jump table will be changed to match. If your machine language routines only use the system ROM routines through the KERNAL, it will take much less work to modify them. The KERNAL is the operating system of the VIC computer. All input, output, and memory management are controlled by the KERNAL.

To simplify the machine language program you write, and to make sure that future versions of the VIC operating system don't make your machine language programs obsolete, the KERNAL contains a jump table for you to use. By taking advantage of the 39 input/output routines and other utilities accessible from the table, not only will you save time, but you will also make it easier to translate your programs from one Commodore computer to another.

The jump table is located on the last page of memory, in read-only memory.

To use the KERNAL jump table, first you set up the parameters that the KERNAL routine needs to work. Then JSR to the proper place in the KERNAL jump table. After performing its function, the KERNAL transfers control back to your machine language program. Depending on which KERNAL routine you are using, certain registers may pass parameters back to your program. The particular registers for each KERNAL routine may be found in the individual descriptions of KERNAL subroutines.

A good question at this point is why use the jump table at all? Why not just JSR directly to the KERNAL subroutine involved? The jump table is used so that if the KERNAL or BASIC is changed, your machine language programs will still work. In future operating systems the routines may be moved in memory . . . but the jump table will still work correctly!

HOW TO USE THE KERNAL

When writing machine language programs it is often convenient to use the routines which are already part of the operating system for input/output, access to the system clock, memory management, and similar operations. It is an unnecessary duplication of effort to write these routines again, so easy access to the operating system helps speed machine language programming.

As mentioned before, the KERNAL is a jump table. This is just a collection of JMP instructions to many operating system routines.

To use a KERNAL routine you must first make all preparations that the routine demands . . . if the routine says that you must have called another KERNAL routine first, then that routine must be called. If the routine expects you to put a number in the accumulator, then that number must be there. Otherwise your routines have little chance of working the way you expect them to work.

After all preparations are made, you must call the routine by means of the JSR instruction. All KERNAL routines you can access are structured as SUBROUTINES, ending with an RTS instruction. When the KERNAL routine has finished its task, control will be returned to your program at the instruction after the JSR.

Many of the KERNAL routines return error codes in the status word or the accumulator in case of problems. Good programming practice and the success of your machine language programs demand that you handle this properly. If you ignore an error return, the rest of your program might bomb.

That's all there is in using the KERNAL—these three steps—

1. Set up
2. Call the routine
3. Error handling

The following conventions are used in describing the KERNAL routines.

FUNCTION NAME: Name of the KERNAL routine.

CALL ADDRESS: This is the call address of the KERNAL routine, given in hexadecimal.

COMMUNICATION REGISTERS: Registers listed under this heading are used to pass parameters to and from the KERNAL routines.

PREPARATORY ROUTINES: Certain KERNAL routines require that data be set up before they can operate. The routines needed are listed here.

ERROR RETURNS: A return from a KERNAL routine with the CARRY set indicates that an error was encountered in processing. The accumulator will contain the number of the error.

STACK REQUIREMENTS: This is the actual number of stack bytes used by the KERNAL routine.

REGISTERS AFFECTED: All registers used by the KERNAL routine are listed here.

DESCRIPTION: A short tutorial on the function of the KERNAL routine is given here.

The list of the KERNAL routines follows.

USER CALLABLE KERNAL ROUTINES

NAME	ADDRESS	FUNCTION
	HEX DECIMAL	
ACPTR	\$FFA5 65445	Input byte from serial port
CHKIN	\$FFC6 65478	Open channel for input
CHKOUT	\$FFC9 65481	Open channel for output
CHRIN	\$FFCF 65487	Input character from channel
CHROUT	\$FFD2 65490	Output character to channel
CIOUT	\$FFA8 65448	Output byte to serial port
CLALL	\$FFE7 65511	Close all channels and files
CLOSE	\$FFC3 65475	Close a specified logical file
CLRCHN	\$FFCC 65484	Close input and output channels
GETIN	\$FFE4 65512	Get character from keyboard queue (keyboard buffer)
IOBASE	\$FFF3 65523	Returns base address of I/O devices
LISTEN	\$FFB1 65457	Command devices on the serial bus to LISTEN
LOAD	\$FFD5 65493	Load RAM from a device
MEMBOT	\$FF9C 65436	Read/set the bottom of memory
MEMTOP	\$FF99 65433	Read/set the top of memory
OPEN	\$FFC0 65472	Open a logical file
PLOT	\$FFF0 65520	Read/set X,Y cursor position
RDTIM	\$FFDE 65502	Read real time clock
READST	\$FFB7 65463	Read I/O status word
RESTOR	\$FF8A 65415	Restore default I/O vectors
SAVE	\$FFD8 65496	Save RAM to device
SCNKEY	\$FF9F 65439	Scan keyboard
SCREEN	\$FFED 65517	Return X,Y organization of screen

NAME	ADDRESS	FUNCTION
	HEX DECIMAL	
SECOND	\$FF93 65427	Send secondary address after LISTEN
SETLFS	\$FFBA 65466	Set logical, first, and second addresses
SETMSG	\$FF90 65424	Control KERNAL messages
SETNAM	\$FFBD 65469	Set filename
SETTIM	\$FFDB 65499	Set real time clock
SETTMO	\$FFA2 65442	Set timeout on serial bus
STOP	\$FFE1 65505	Scan stop key
TALK	\$FFB4 65460	Command serial bus device to TALK
TKSA	\$FF96 65430	Send secondary address after TALK
UDTIM	\$FFEA 65514	Increment real time clock
UNLSN	\$FFAE 65454	Command serial bus to UNLISTEN
UNTLK	\$FFAB 65451	Command serial bus to UNTALK
VECTOR	\$FF84 65412	Read/set vectored I/O

B-1. Function name: ACPTR

Purpose: Get data from the serial bus

Call address: \$FFA5

Communication registers: .A

Preparatory routines: TALK ,TKSA

Error returns: See READST

Stack requirements: 13

Registers affected: .A, .X

Description: This is the routine to use to get information from a device on the serial bus (like the disk). This routine gets a byte of data off the serial bus using full handshaking. The data is returned in the accumulator. To prepare for this routine the TALK routine must have been called first to command the device on the serial bus to send data on the bus. If the input device needs a secondary command, it must be sent by using the TKSA KERNAL routine before calling this routine. Errors are returned in the status word. The READST routine is used to read the status word.

To use this routine:

0) Command a device on the serial bus to prepare to send data to the VIC.

(Use the TALK and TKSA kernal routines).

1) Call this routine (using JSR)

2) Store or otherwise use the data.

EXAMPLE:

Get a byte from the bus

- 1) JSR ACPTR
- 2) STA DATA

B-2. Function name: CHKIN

Purpose: Open a channel for input

Call address: \$FFC6

Communication registers: .X

Preparatory routines: (OPEN)

Error returns: 3,5,6

Stack requirements: None

Registers affected: .A, .X

Description: Any logical file that has already been opened by the KERNAL OPEN routine can be defined as an input channel by this routine. Naturally, the device on the channel must be an input device. Otherwise, an error will occur, and the routine will abort.

If you are getting data from anywhere other than the keyboard, this routine must be called before using either the CHRIN or the GETIN KERNAL routines for data input. If input from the keyboard is desired, and no other input channels are opened, then the calls to this routine, and to the OPEN routine, are not needed.

When this routine is used with a device on the serial bus, this routine automatically sends the talk address (and the secondary address if one was specified by the OPEN routine) over the bus.

To use this routine:

- 0) OPEN the logical file (if necessary; see description above).
- 1) Load the .X register with number of the logical file to be used.
- 2) Call this routine (using a JSR command).

Possible errors are:

#3: File not open

#5: Device not present

#6: File not an input file

EXAMPLE:

; PREPARE FOR INPUT FROM LOGICAL FILE 2

- 1) LDX #2
- 2) JSR CHKIN

B-3. Function name: CHKOUT

Purpose: Open a channel for output

Call address: \$FFC9

Communication registers: .X

Preparatory routines: (OPEN)

Error returns: 3,5,7

Stack requirements: None

Registers Affected: .A, .X

Description: Any logical file number which has been created by the KERNAL routine OPEN can be defined as an output channel. Of course, the device you intend opening a channel to must be an output device. Otherwise, an error will occur, and the routine will be aborted.

This routine must be called before any data is sent to any output device unless you want to use the VIC screen as your output device. If screen output is desired, and there are no other output channels already defined, then the calls to this routine, and to the OPEN routine are not needed.

When used to open a channel to a device on the serial bus, this routine will automatically send the LISTEN address specified by the OPEN routine (and a secondary address if there was one).

How to use: Remember: this routine is NOT NEEDED to send data to the screen. 0) Use the KERNAL OPEN routine to specify a logical file number, a LISTEN address, and a secondary address (if needed).

1) Load the .X register with the logical file number used in the open statement.

2) Call this routine (by using the JSR instruction).

;DEFINE LOGICAL FILE 3 AS AN OUTPUT CHANNEL

1) LDX #3

2) JSR CHKOUT

Possible error returns:

3: File not open

5: Device not present

7: Not an output file

B-4. Function name: CHRIN

Purpose: Get a character from the input channel

Call address: \$FFCF

Communication registers: .A

Preparatory routines: (OPEN, CHKIN)

Error returns: See READST

Stack requirements: None

Registers affected: .A, .X

Description: This routine will get a byte of data from the channel already set up as the input channel by the KERNAL routine CHKIN.

If the CHKIN has not been used to define another input channel, data is expected from the keyboard. The data byte is returned in the accumulator. The channel remains open after the call.

Input from the keyboard is handled in a special way. First, the cursor is turned on, and will blink until a carriage return is typed on the keyboard. All characters on the line (up to 88 characters) will be stored in the BASIC input buffer. Then the characters can be retrieved one at a time by calling this routine once for each character. When the carriage return is retrieved, the entire line has been processed. The next time this routine is called, the whole process begins again, i.e., by flashing the cursor.

How to use:

FROM THE KEYBOARD

- 1) Call this routine (using the JSR instruction).
- 2) Retrieve a byte of data by calling this routine.
- 3) Store the data byte.
- 4) Check if it is the last data byte (is it a CR ?). If not, go to step 2.

EXAMPLE:

- | | | |
|----|--------------|-------------------------------------|
| 1) | LDX \$00 | ;Store 00 in the .X register |
| 2) | RD JSR CHRIN | |
| | STA DATA,X | ;store the Xth data byte in the Xth |
| | INX | ;location in the data area. |
| 3) | CMP #CR | ;Is it a carriage return? |
| 4) | BNE RD | ;no, get another data byte |

EXAMPLE:

```
JSR CHRIN
STA DATA
```

FROM OTHER DEVICES

- 0) Use the KERNAL OPEN and CHKIN routines.
- 1) Call this routine (using a JSR instruction).
- 2) Store the data.

EXAMPLE:

```
JSR CHRIN
STA DATA
```

B-5. Function name: CHROUT

Purpose: Output a character

Call address: \$FFD2

Communication registers: .A
Preparatory routines: (CHKOUT, OPEN)
Error returns: See READST
Stack requirements: None
Registers affected: None

Description: This routine will output a character to an already opened channel. Use the KERNAL OPEN and CHKOUT routines to set up the output channel before calling this routine. If this call is omitted, data will be sent to the default output device (number 3, on the screen). The data byte to be output is loaded into the accumulator, and this routine is called. The data is then sent to the specified output device. The channel is left open after the call.

NOTE: Care must be taken when using this routine to send data to a serial device since data will be sent to all open output channels on the bus. Unless this is desired, all open output channels on the serial bus other than the actually intended destination channel must be closed by a call to the KERNAL close channel routine.

How to use:

0) Use the CHKOUT KERNAL routine if needed (see description above).

- 1) Load the data to be output into the accumulator.
- 2) Call this routine.

EXAMPLE:

```
                                ;Duplicate the BASIC instruction CMD 4, "A";  
LDX #4                        ;LOGICAL FILE #4  
JSR CHKOUT ;OPEN CHANNEL OUT  
LDA #'A  
JSR CHROUT ;SEND CHARACTER
```

B-6. Function name: CIOUT

Purpose: Transmit a byte over the serial bus
Call address: \$FFA8
Communication registers: .A
Preparatory routines: LISTEN, [SECOND]
Error returns: See READST
Stack requirements: None
Registers affected: .A

Description: This routine is used to send information to devices on the serial bus. A call to this routine will put a data byte onto the serial bus using full serial handshaking. Before this routine is called, the LISTEN KERNAL routine must be used to command a device

on the serial bus to get ready to receive data. (If a device needs a secondary address, it must also be sent by using the SECOND KERNAL routine.)

The accumulator is loaded with a byte to handshake as data on the serial bus. A device must be listening or the status word will return a timeout. This routine always buffers one character. (The routine holds the previous character to be sent back.) So when a call to the KERNAL UNLSN routine is made to end the data transmission, the buffered character is sent with EOI set. Then the UNLSN command is sent to the device.

How to use:

0) Use the LISTEN KERNAL routine (and the SECOND routine if needed).

- 1) Load the accumulator with a byte of data.
- 2) Call this routine to send the data byte.

EXAMPLE:

```
;Send an X to the serial bus  
LDA #'X  
JSR CIOUT
```

B-7. Function name: CLALL

Purpose: Close all files
Call address: \$FFE7
Communication registers: None
Preparatory routines: None
Error returns: None
Stack requirements: 11
Registers affected: .A, .X

Description: This routine closes all open files. When this routine is called, the pointers into the open file table are reset, closing all files. Also, the routine automatically resets the I/O channels.

How to use:

- 1) Call this routine.

EXAMPLE:

```
;USED AT START OF EXECUTION FOR INITIALIZATION  
JSR CLRCHN ;CLOSE FILES  
JMP RUN ;BEGIN EXECUTION
```

B-8. Function name: CLOSE

Purpose: Close a logical file
Call address: \$FFC3

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: None

Registers affected: .A, .X

Description: This routine is used to close a logical file after all I/O operations have been completed on that file. This routine is called after the accumulator is loaded with the logical file number to be closed (the same number used when the file was opened using the OPEN routine).

How to use:

- 1) Load the accumulator with the number of the logical file to be closed.
- 2) Call this routine.

EXAMPLE:

```
;CLOSE 15  
LDA #15  
JSR CLOSE
```

B-9. Function name: CLRCHN

Purpose: Clear I/O channels

Call address: \$FFCC

Communication registers: None

Preparatory routines: None

Error routines: None

Stack requirements: 9

Registers affected: .A, .X

Description: This routine is called to clear all open channels and restore the I/O channels to their original default values. It is usually called after opening other I/O channels (like to the disk or tape drive) and using them for input/output operations. The default input device is 0 (keyboard). The default output device is 3 (the VIC screen).

If one of the channels to be closed is to the serial port, an UNTALK signal is sent first to clear the input channel or an UNLISTEN is sent to clear the output channel. By not calling this routine (and leaving listener(s) active on the serial bus) several devices can receive the same data from the VIC at the same time. One way to take advantage of this would be to command the printer to TALK and the disk to LISTEN. This would allow direct printing of a disk file.

How to use:

- 1) Call this routine using the JSR instruction

EXAMPLE:

JSR CLRCHN

B-10. Function name: GETIN

Purpose: Get a character from the keyboard buffer

Call address: \$FFE4

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: None

Registers affected: .A, .X

Description: This subroutine removes one character from the keyboard queue and returns it as an ASCII value in the accumulator. If the queue is empty, the value returned in the accumulator will be zero. Characters are put into the queue automatically by an interrupt driven keyboard scan routine which calls the SCNKEY routine. The keyboard buffer can hold up to ten characters. After the buffer is filled, additional characters are ignored until at least one character has been removed from the queue.

How to use:

- 1) Call this routine using a JSR instruction
- 2) Check for a zero in the accumulator (empty buffer)
- 3) Process the data

EXAMPLE:

```
;WAIT FOR A CHARACTER  
WAIT JSR GETIN  
CMP #0  
BEQ WAIT
```

B-11. Function name: IOBASE

Purpose: Define I/O memory page

Call address: \$FFF3

Communication registers: X,Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .X, .Y

Description: This routine will set the X and Y registers to the address of the memory section where the memory mapped I/O devices are located. This address can then be used with an offset to

access the memory mapped I/O devices in the VIC. The offset will be the number of locations from the beginning of the page that the desired I/O register is located. The .X register will contain the low order address byte, while the .Y register will contain the high order address byte.

This routine exists to provide compatibility between the VIC 20 and future models of the VIC. IF the I/O locations for a machine language program are set by a call to this routine, they should still remain compatible with future versions of the VIC, the KERNAL and BASIC.

How to use:

- 1) Call this routine by using the JSR instruction.
- 2) Store the .X and the .Y registers in consecutive locations.
- 3) Load the .Y register with the offset.
- 4) Access that I/O location.

EXAMPLE:

; Set the data direction register of the user port to 0 (input)

- 1) JSR IOBASE
- 2) STX POINT ;set base registers
STY POINT + 1
- 3) LDY #2
- 4) LDA #0 ;offset for DDR of the user port
STA (POINT)Y ;Set DDR to 0

B-12. Function name: LISTEN

Purpose: Command a device to LISTEN

Call Address: \$FFB1

Communication registers: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: None

Registers affected: .A

Description: This routine will command a device on the serial bus to receive data. The accumulator must be loaded with a device number between 4 and 31 before calling the routine. LISTEN will OR the number bit by bit to convert to a listen address, then transmit this data as a command on the serial bus. The specified device will then go into listen mode, and be ready to accept information.

How to use:

- 1) Load the accumulator with the number of the device to command to LISTEN.
- 2) Call this routine using the JSR instruction.

EXAMPLE:

```
;COMMAND DEVICE #8 TO LISTEN
LDA #8
JSR LISTEN
```

B-13. Function name: LOAD

Purpose: Load RAM from device
Call address: \$FFD5
Communication registers: .A, .X, .Y
Preparatory routines: SETLFS, SETNAM
Error returns: 0,4,5,8,9
Stack requirements: None
Registers affected: .A, .X, .Y

Description: This routine will load data bytes from any input device directly into the memory of the VIC. It can also be used for a verify operation, comparing data from a device with the data already in memory, leaving the data stored in RAM unchanged. The accumulator (.A) must be set to 0 for a load operation, or 1 for a verify. If the input device was OPENed with a secondary address (SA) of 0, the header information from device will be ignored. In this case, the .X and .Y registers must contain the starting address for the load. If the device was addressed with a secondary address of 0,1, or 2 the data will load into memory starting at the location specified by the header. This routine returns the address of the highest RAM location which was loaded.

Before this routine can be called, the KERNAL SETLFS, and SETNAM routines must be called.

How to use

0) Call the SETLFS, and SETNAM routines. If a relocated load is desired, use the SETLFS routine to send a secondary address of 3.

- 1) Set the .A register to 0 for load, 1 for verify.
- 2) If a relocated load is desired, the .X and .Y registers must be set to the start address for the load.
- 3) Call the routine using the JSR instruction.

EXAMPLE:

```
;LOAD A FILE FROM TAPE
```

```
0)   LDA  #DEVICE1      ;set device number
      LDX  #FILENO       ;set logical file number
      LDY  CMD1          ;set secondary address
      JSR  SETLFS
      LDA  #NAME1 - NAME ;load .A with number of char-
                           ;acters
                           ;in filename
```

```

        LDX  #<NAME      ;Load .X and .Y with address of
        LDY  #>NAME      ;filename
        JSR  SETNAM
1)      LDA  #0           ;set flag for a load
2)      LDX  #$FF        ;default start
        LDY  #$FF
3)      JSR  LOAD
        STX  VARTAB      ;end of load
        STY  VARTAB + 1
        JMP  START
NAME .BYT 'FILE NAME'
NAME 1;

```

B-14. Function name: MEMBOT

Purpose: Set bottom of memory

Call address: \$FF9C

Communication registers: .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: None

Registers affected: .X, .Y

Description: This routine is used to set the bottom of the memory. If the accumulator carry bit is set when this routine is called, a pointer to the lowest byte of RAM will be returned in the .X and .Y registers. On the unexpanded VIC the initial value of this pointer is \$1000. If the accumulator carry bit is clear (= 0) when this routine is called, the values of the .X and .Y registers will be transferred to the low and high bytes respectively of the pointer to the beginning of RAM.

How to use:

TO READ THE BOTTOM OF RAM

- 1) Set the carry.
- 2) Call this routine.

TO SET THE BOTTOM OF MEMORY

- 1) Clear the carry.
- 2) Call this routine.

EXAMPLE:

```

; MOVE BOTTOM OF MEMORY UP 1 PAGE
SEC          ;READ MEMORY BOTTOM
JSR MEMBOT
INY
CLC          ;SET MEMORY BOTTOM TO NEW VALUE
JSR MEMBOT

```

B-15. Function name: MEMTOP

Purpose: Set the top of RAM

Call address: \$FF99

Communication registers: .X,.Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .X, .Y

Description: This routine is used to set the top of RAM. When this routine is called with the carry bit of the accumulator set, the pointer to the top of RAM will be loaded into the .X and .Y registers. When this routine is called with the accumulator carry bit clear, the contents of the .X and .Y registers will be loaded in the top of memory pointer, changing the top of memory.

EXAMPLE:

```
;DEALLOCATE THE RS-232 BUFFER  
SEC  
JSR MEMTOP ;READ TOP OF MEMORY  
DEX  
CLC  
JSR MEMTOP ;SET NEW TOP OF MEMORY
```

B-16. Function name: OPEN

Purpose: Open a logical file

Call address: \$FFC0

Communication registers: None

Preparatory routines: SETLFS, SETNAM

Error returns: 1,2,4,5,6

Stack requirements: None

Registers affected: .A, .X, .Y

Description: This routine is used to open a logical file. Once the logical file is set up, it can be used for input/output operations. Most of the I/O KERNAL routines call on this routine to create the logical files to operate on. No arguments need to be set up to use this routine, but both the SETLFS and SETNAM KERNAL routines must be called before using this routine.

How to use:

- 0) Use the SETLFS routine.
- 1) Use the SETNAM routine.
- 2) Call this routine.

EXAMPLE:

This is an implementation of the BASIC statement: OPEN 15,8,15,"I/O"

```
LDA #NAME2 - NAME      ;LENGTH OF FILE NAME FOR
                        SETLFS
LDY #>NAME
JSR SETNAM
    LDA #15
    LDX #8
    LDY #15
    JSR SETLFS
    JSR OPEN
NAME .BYT 'I/O'
NAME2
```

B-17. Function name: PLOT

Purpose: Set cursor location
Call address: \$FFF0
Communication registers: .A,X,Y
Preparatory routines: None
Error returns: None
Stack requirements: 2
Registers affected: .A, .X, .Y

Description: A call to this routine, with the accumulator carry flag set, loads the current position of the cursor on the screen (in X,Y coordinates) into the .X and .Y registers. X is the column number of the cursor location (0-21), and Y is the row number of the location of the cursor (0-22). A call with the carry bit clear moves the cursor to X,Y as determined by the .X and .Y registers.

How to use:

READING CURSOR LOCATION

- 1) Set the carry flag.
- 2) Call this routine.
- 3) Get the X and Y position from the .X and .Y registers respectively.

SETTING CURSOR LOCATION

- 1) Clear carry flag.
- 2) Set the .X and .Y registers to the desired cursor location.
- 3) Call this routine.

EXAMPLE:

```
; MOVE THE CURSOR TO 5,5  
LDX #5  
LDY #5  
CLC  
JSR PLOT
```

B-18. Function name: RDTIM

Purpose: Read system clock

Call address: \$FFDE

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A, .X, .Y

Description: This routine is used to read the system clock. The clock's resolution is a 60th of a second. Three bytes are returned by the routine. The accumulator contains the most significant byte, the X index register contains the next most significant byte, and the Y index register contains the least significant byte.

EXAMPLE:

```
JSR RDTIM  
STY TIME  
STX TIME + 1  
STA TIME + 2  
...  
TIME * = * + 3
```

B-19. Function name: READST

Purpose: Read status word

Call address: \$FFB7

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A

Description: This routine returns the current status of the I/O devices in the accumulator. The routine is usually called after new communication to an I/O device. The routine will give information about device status, or errors that have occurred during the I/O operation.

The bits returned in the accumulator contain the following information (see table below):

How to use:

- 1) Call this routine.
- 2) Decode the information in the .A register as it refers to your program.

EXAMPLE:

```
; CHECK FOR END OF FILE DURING READ
JSR READST
AND #64      ;check eof bit
BNE EOF      ;branch on eof
```

ST Bit Position	ST Numeric Value	Cassette Read	Serial/RW	Tape Verify + Load
0	1		Time out	
write				
1	2		Time out	
read				
2	4	Short block		Short block
3	8	Long block		Long block
4	16	Unrecoverable read error		Any mismatch
5	32	Checksum error		Checksum error
6	64	End of file	EOI line	
7	-128	End of tape	Device not present	End of tape

B-20. Function name: RESTOR

Purpose: Restore default system and interrupt vectors

Call address: \$FF8A

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A, .X, .Y

Description: This routine restores the default values of all system vectors used in KERNAL and BASIC routines and interrupts. (See appendix D for the default vector contents). The KERNAL VECTOR routine is used to read and alter individual system vectors.

How to use:

- 1) Call this routine.

EXAMPLE:

JSR RESTOR

B-21. Function name: SAVE

Purpose: Save memory to a device

Call address: \$FFD8

Communication registers: .A, .X, .Y

Preparatory routines: SETLFS, SETNAM

Error returns: 5,8,9

Stack requirements: None

Registers affected: .A, .X, .Y

Description: This routine saves a section of memory. Memory is saved from an indirect address on page 0 specified by the accumulator to the address stored in the .X and .Y registers to a logical file (an input/output device). The SETLFS and SETNAM routines must be used before calling this routine. However, a file name is not required to SAVE to device 1 (the cassette tape recorder). Any attempt to save to other devices without using a file name results in an error.

NOTE: Device 0 (the keyboard) and device 3 (the screen) cannot be SAVED to. If the attempt is made, an error will occur, and the SAVE stopped.

How to use:

0) Use the SETLFS routine and the SETNAM routine (unless a SAVE with no file name is desired on a save to the tape recorder).

1) Load two consecutive locations on page 0 with a pointer to the start of your save (in standard 6502 low byte first, high byte next format).

2) Load the accumulator with the single byte page zero offset to the pointer.

3) Load the .X and .Y registers with the low byte and high byte respectively of the location of the end of the save.

4) Call this routine.

EXAMPLE:

```
LDA #1          ;DEVICE = 1: CASSETTE
                JSR SETLFS
LDA #0          ;NO FILE NAME
JSR SETNAM
LDA PROG        ;LOAD START ADDRESS OF SAVE
STA TXTTAB      ; (LOW BYTE)
LDA PROG + 1
                STA TXTTAB + 1          (HIGH BYTE)
```

```

LDX VARTAB ;Load .X WITH LOW BYTE OF END OF SAVE
LDY VAR
TAB+1      ;      .Y WITH HIGH BYTE
LDA
#<TXTTAB   ;LOAD ACCUMLATOR WITH PAGE 0 OFF-
SET
JSR SAVE

```

B-22. Function name: SCNKEY

Purpose: Scan the keyboard

Call address: \$FF9F

Communication registers: None

Preparatory routines: None

Error returns: None

Stack requirements: None

Registers affected: .A, .X, .Y

Description: This routine will scan the VIC keyboard and check for pressed keys. It is the same routine called by the interrupt handler. If a key is down, its ASCII value is placed in the keyboard queue.

How to use:

- 1) Call this routine

EXAMPLE:

```

GET  JSR SCNKEY      ;SCAN KEYBOARD
      JSR GETIN       ;GET CHARACTER
      CMP #0          ;IS IT NULL?
      BEQ GET         ;YES. . .SCAN AGAIN
      JSR CHROUT      ;PRINT IT

```

B-23. Function name: SCREEN

Purpose: Return screen format

Call address: \$FFED

Communication registers: X, Y

Preparatory routines: None

Stack requirements: 2

Registers affected: .X, .Y

Description: This routine returns the format of the screen, e.g., 22 columns in .X and 23 lines in .Y. This routine can be used to determine what machine a program is running on, and has been implemented on the VIC to help upward compatibility in programs.

How to use:

- 1) Call this routine.

EXAMPLE:

```

JSR SCREEN

```

STX MAXCOL
STY MAXROW

B-24. Function name: SECOND

Purpose: Send secondary address for LISTEN

Call address: \$FF93

Communication registers: .A

Preparatory routines: LISTEN

Error returns: See READST

Stack requirements: None

Registers affected: .A

Description: This routine is used to send a secondary address to an I/O device after a call to the LISTEN routine is made, and the device commanded to LISTEN. The routine cannot be used to send a secondary address after a call to the TALK routine.

A secondary address is usually used to give set-up information to a device before I/O operations begin.

When a secondary address is to be sent to a device on the serial bus, the address must first be ORed with \$60.

How to use:

- 1) Load the accumulator with the secondary address to be sent.
- 2) Call this routine.

EXAMPLE:

```
;ADDRESS DEVICE #8 WITH COMMAND (SECONDARY  
ADDRESS) #15
```

```
LDA #8
```

```
JSR LISTEN
```

```
LDA #15
```

```
ORA #60
```

```
JSR SECOND
```

B-25. Function name: SETLFS

Purpose: Set up a logical file

Call address: \$FFBA

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: None

Description: This routine will set the logical file number, device address, and secondary address (command number) for other KERNAL routines.

The logical file number is used by the system as a key to the file table created by the OPEN file routine. Device addresses can

range from 0 to 30. The following codes are used by the VIC to stand for the following CBM devices.

ADDRESS	DEVICE
0	Keyboard
1	Cassette #1
2	RS-232C device
3	CRT display
4	Serial Bus printer
8	CBM Serial bus disk drive

Device numbers 4 or greater automatically refer to devices on the serial bus.

A command to the device is sent as a secondary address on the serial bus after the device number is sent during the serial attention handshaking sequence. If no secondary address is to be sent, the .Y index register should be set to 255.

How to use:

- 1) Load the accumulator with the logical file number.
- 2) Load the .X index register with the device number.
- 3) Load the .Y index register with the command.

EXAMPLE:

For logical file 32, device #4, and no command:

```
LDA #32
LDX #4
LDY #255
JSR SETLFS
```

B-26. Function name: SETMSG

Purpose: Control system message output

Call address: \$FF90

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A

Description: This routine controls the printing of error and control messages by the KERNAL. Either print error messages or print control messages can be selected by setting the accumulator when the routine is called. FILE NOT FOUND is an example of an error message. PRESS PLAY ON CASSETTE is an example of a control message.

Bits 6 and 7 of this value determine where the message will come from. If bit 7 is 1, one of the error messages from the KERNAL will be printed. If bit 6 is set, a control message will be printed.

How to use:

- 1) Set accumulator to desired value.
- 2) Call this routine.

EXAMPLE:

```
LDA #$40
JSR SETMSG ;TURN ON CONTROL MESSAGES
LDA #$80 ;
JSR SETMSG TURN ON ERROR MES-
SAGES

LDA #0
JSR SETMSG ;TURN OFF ALL KERNAL MESSAGES
```

B-27. Function name: SETNAM

Purpose: Set up file name

Call address: \$FFBD

Communication registers: .A, .X, .Y

Preparatory routines: None

Stack requirements: None

Registers affected: None

Description: This routine is used to set up the file name for the OPEN, SAVE, or LOAD routines. The accumulator must be loaded with the length of the file name. The .X and .Y registers must be loaded with the address of the file name, in standard 6502 low byte, high byte format. The address can be any valid memory address in the system where a string of characters for the file name is stored. If no file name is desired, the accumulator must be set to 0, representing a zero file length. The .X and .Y registers may be set to any memory address in that case.

How to use:

- 1) Load the accumulator with the length of the file name.
- 2) Load the .X index register with the low order address of the file name.
- 3) Load the .Y index register with the high order address.
- 4) Call this routine.

EXAMPLE:

```
LDA #NAME2-NAME ;LOAD LENGTH OF FILE NAME
LDX #<NAME
LDY #>NAME
JSR SETNAM
```

B-28. Function name: SETTIM

Purpose: Set the system clock

Call address: \$FFDB

Communication registers: .A, .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: None

Description: A system clock is maintained by an interrupt routine that updates the clock every 1/60th of a second (one 'jiffy'). The clock is three bytes long, which gives it the capability to count up to 5,184,000 jiffies (24 hours). At that point the clock resets to zero. Before calling this routine to set the clock, the accumulator must contain the most significant byte, the .X index register the next most significant byte, and the .Y index register the least significant byte of the initial time setting (in jiffies).

How to use:

- 1) Load the accumulator with the MSB of the 3 byte number to set the clock.
- 2) Load the .X register with the next byte.
- 3) Load the .Y register with the LSB.
- 4) Call this routine.

EXAMPLE:

```
;SET THE CLOCK TO 10 MINUTES = 3600 JIFFIES
LDA #0          ; MOST SIGNIFICANT
LDX #>3600
LDY #<3600      ; LEAST SIGNIFICANT
JSR SETTIM
```

B-29. Function name: SETTMO

Purpose: Set serial bus timeout flag

Call address: \$FFA2

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: None

Description: This routine sets the timeout flag for the serial bus. When the timeout flag is set, the VIC will wait for a device on the serial port for 64 milliseconds. If the device does not respond to the VIC's DAV signal within that time the VIC will recognize an error condition and leave the handshake sequence. When this routine is called when the accumulator contains a 0 in bit 7, timeouts are enabled. A 1 in bit 7 will disable the timeouts. NOTE: The VIC uses the timeout feature to communicate that a disk file is not found on an attempt to OPEN a file.

How to use:**TO SET THE TIMEOUT FLAG**

- 1) Set bit 7 of the accumulator to 0.
- 2) Call this routine.

TO RESET THE TIMEOUT FLAG

- 1) Set bit 7 of the accumulator to 1.
- 2) Call this routine.

EXAMPLE:

```
;DISABLE TIMEOUT  
LDA #0  
JSR SYSTMO
```

B-30. Function name: STOP

Purpose: Check if stop key is pressed

Call address: \$FFE1

Communication registers: .A

Preparatory routines: None

Error returns: None

Stack requirements: None

Registers affected: .A, .X

Description: If the STOP key on the keyboard is pressed when this routine is called, the Z flag will be set. All other flags remain unchanged. If the STOP key is not pressed then the accumulator will contain a byte representing the last row of the keyboard scan. The user can also check for certain other keys this way.

How to use this routine:

- 1) Call this routine.
- 2) Test for the zero flag.

EXAMPLE:

```
JSR STOP  
BNE *+5 ;KEY NOT DOWN  
JMP READY ;= . . .STOP
```

B-31. Function name: TALK

Purpose: Command a device on the serial bus to TALK

Call address: \$FFB4

Communication registers: .A

Preparatory routines: None

Error returns: See READST

Stack requirements: None

Registers affected: .A

Description: To use this routine the accumulator must first be loaded with a device number between 4 and 30. When called, this routine then ORs bit by bits to convert this device number to a talk address. Then this data is transmitted as a command on the Serial bus.

How to use:

- 0)
- 1) Load the accumulator with the device number.
- 2) Call this routine.

EXAMPLE:

```
;COMMAND DEVICE #4 TO TALK
LDA #4
JSR TALK
```

B-32. Function name: TKSA

Purpose: Send a secondary address to a device commanded to TALK

Call address: \$FF96
Communication registers: .A
Preparatory routines: TALK
Error returns: See READST
Stack requirements: None
Registers affected: .A

Description: This routine transmits a secondary address on the serial bus for a TALK device. This routine must be called with a number between 4 and 31 in the accumulator. The routine will send this number as a secondary address command over the serial bus. This routine can only be called after a call to the TALK routine. It will not work after a LISTEN.

How to use:

- 0) Use the TALK routine.
- 1) Load the accumulator with the secondary address.
- 2) Call this routine.

EXAMPLE:

```
;TELL DEVICE #4 TO TALK WITH COMMAND #7
LDA #4
JSR TALK
LDA #7
JSR TALKSA
```

B-33. Function name: UDTIM

Call address: \$FFEA

Purpose: Update the system clock

Communication registers: None

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A, .X

Description: This routine updates the system clock. Normally this routine is called by the normal KERNAL interrupt routine every 1/60th of a second. If the user program processes its own interrupts this routine must be called to update the time. Also, the STOP key routine must be called, if the stop key is to remain functional.

How to use:

- 1) Call this routine.

EXAMPLE:

JSR UDTIM

B-34. Function name: UNLSN

Purpose: Send an UNLISTEN command

Call address: \$FFAE

Communication registers: None

Preparatory routines: None

Error returns: See READST

Stack requirements: None

Registers affected: .A

Description: This routine commands all devices on the serial bus to stop receiving data from the VIC. (i.e., UNLISTEN). Calling this routine results in an UNLISTEN command being transmitted on the serial bus. Only devices previously commanded to listen will be affected. This routine is normally used after the VIC is finished sending data to external devices. Sending the UNLISTEN will command the listening devices to get off the serial bus so it can be used for other purposes.

How to use:

- 1) Call this routine.

EXAMPLE:

JSR UNLSN

B-35. Function name: UNTLK

Purpose: Send an UNTALK command

Call address: \$FFAB

Communication registers: None

Preparatory routines: None

Error returns: See READST

Stack requirements: None

Registers affected: .A

Description: This routine will transmit an UNTALK command on the serial bus. All devices previously set to TALK will stop sending data when this command is received.

How to use:

- 1) Call this routine.

EXAMPLE:

```
JSR UNTALK
```

B-36. Function name: VECTOR

Purpose: Manage RAM vectors

Call address: \$FF8D

Communication registers: .X, .Y

Preparatory routines: None

Error returns: None

Stack requirements: 2

Registers affected: .A, .X, .Y

Description: This routine manages all system vector jump addresses stored in RAM. Calling this routine with the accumulator carry bit set will store the current contents of the RAM vectors in a list pointed to by the .X and .Y registers. When this routine is called with the carry clear, the user list pointed to by the .X and .Y registers is transferred to the system RAM vectors. NOTE: This routine requires caution in its use. The best way to use it is to first read the entire vector contents into the user area, alter the desired vectors, and then copy the contents back to the system vectors.

How to use:

READ THE SYSTEM RAM VECTORS

- 1) Set the carry.
- 2) Set the .X and .Y registers to the address to put the vectors.
- 3) Call this routine.

LOAD THE SYSTEM RAM VECTORS

- 1) Clear the carry bit.
- 2) Set the .X and .Y registers to the address of the vector list in RAM that must be loaded
- 3) Call this routine.

EXAMPLE:

CHANGE THE INPUT ROUTINES TO NEW SYSTEM

```
LDX #<USER
LDY #>USER
SEC
JSR VECTOR    ;read old vectors
LDA #<MYINP   ;change input
STA USER+10
LDA #>MYINP
STA USER+11
LDX #<USER
LDY #>USER
CLC
JSR VECTOR    ;alter system
```

...
USER *= * + 26

ERROR CODES

The following is a list of error messages which can occur when using the KERNAL routines. If an error occurs during a KERNAL routine, the carry bit of the accumulator is set, and the number of the error message is returned in the accumulator.

NUMBER	MEANING
0	Routine terminated by the STOP key
1	Too many open files
2	File already open
3	File not open
4	File not found
5	Device not present
6	File is not an input file
7	File is not an output file
8	File name is missing
9	Illegal device number

KERNAL POWER UP ACTIVITIES

1) KERNAL checks for the presence of ROM at \$A000

The KERNAL looks in memory at \$A000 for the AUTO START ROM SEQUENCE. If this sequence is present, control is transferred to the ROM program.

If this AUTO START ROM code is not present normal system initialization continues.

2) RAM test

The RAM TEST routine first clears memory from \$0000-\$00FF and from \$0200-\$03FF. The cassette buffer pointer is initialized to \$033C.

The RAM test starts at location \$0400 and works upward, checking for the first byte of RAM memory (start-of-RAM). If this location is greater than \$1000, then memory is considered bad and an error screen is shown.

Once the test has found the start of RAM it continues, checking upward for the first non-RAM location (top-of-RAM). If this location is <\$2000, then memory is considered bad and an error screen is shown.

If the top-of-RAM location is greater or equal to \$2100, then the screen is set to start at \$1000, the bottom-of-memory is set to \$1200, and the top-of-memory is set to the top-of-RAM location.

If the top-of-RAM location is less than \$2100, then the screen is set to start at \$1E00, the bottom-of-memory is set to start-of-RAM, and the top-of-memory is set to \$1E00.

3) Other Activities

I/O vectors are set to default values.

The indirect jump table in low memory is established.

The GETCHAR routine is created on page zero.

The screen is then cleared, and the 'BYTES FREE' power up message is displayed. Control of the system is turned over to BASIC and the user.

VIC CHIPS

6560 VIDEO INTERFACE CHIP

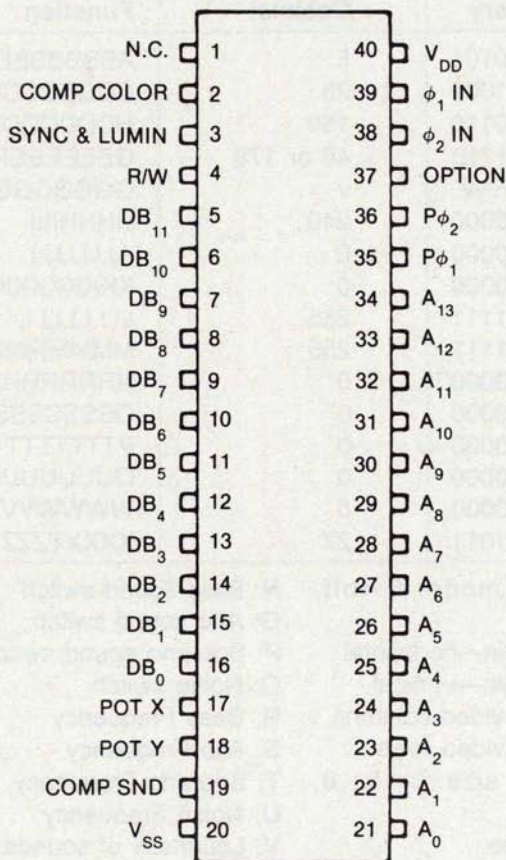
The 6560 Video Interface Chip (VIC) is designed for color video graphics applications such as low cost CRT terminals, biomedical monitors, control system displays and arcade or home video games. It provides all of the circuitry necessary for generating color programmable character graphics with high screen resolution. VIC also incorporates sound effects and A/D converters for use in a video game environment.

FEATURES

- Fully expandable system with a 16K byte address space
- System uses industry standard 8 bit wide ROMS and 4 bit wide RAMS
- Mask programmable sync generation, NTSC-6560, PAL-6561
- On-chip color generation (16 colors)
- Up to 600 independently programmable and movable background locations on a standard TV
- Screen centering capability
- Screen grid size up to 192 Horizontal by 200 Vertical dots
- Two selectable graphic character sizes
- On-chip sound system including:
 - a) Three independent, programmable tone generators
 - b) White noise generator
 - c) Amplitude modulator
- Two on-chip 8 bit A/D converters
- ON-chip DMA and address generation
- No CPU wait states or screen hash during screen refresh
- Interlaced/Non-Interlaced switch
- 16 addressable control registers
- Light gun/pen for target games
- 2 modes of color operation

PIN CONFIGURATION

6560



A: Interlace mode: A normal video frame is sent to the TV 60 times each second. Interlace mode cuts the number of repetitions in half. When used with multiplexing equipment, this allows the VIC picture to be blended with a picture from another source.

To turn off: POKE 36864, PEEK(36864) AND 127

To turn on: POKE 36864, PEEK(36864) OR 128

B: Screen origin—horizontal: This determines the positioning of the image on the TV screen. The normal value is 5. Lowering the value moves the screen to the left, and increasing it moves the image to the right.

To change value: POKE 36864, PEEK(36864) AND 128 OR X

VIC CHIP

LOC Hex	START VALUE-5K VIC		Bit Function
	Binary	Decimal	
9000	00000101	5	ABBBBBBBB
9001	00011001	25	CCCCCCCCC
9002	10010110	150	HDDDDDDDD
9003	v0101110	46 or 176	GEEEEEEEF
9004	vvvvvvvv	v	GGGGGGGGG
9005	11110000	240	HHHHIIII
9006	00000000	0	JJJJJJJJ
9007	00000000	0	KKKKKKKKK
9008	11111111	255	LLLLLLLLL
9009	11111111	255	MMMMMMMMM
900A	00000000	0	NNNNNNNNN
900B	00000000	0	OSSSSSSSS
900C	00000000	0	PTTTTTTTT
900D	00000000	0	QUUUUUUUU
900E	00000000	0	WWWVVVVVV
900F	00011011	27	XXXXYZZZ

- | | |
|--|----------------------------------|
| A: Interlace mode: 0 = off, 1 = on | N: Bass sound switch |
| B: Screen origin—horizontal | O: Alto sound switch |
| C: Screen origin—vertical | P: Soprano sound switch |
| D: Number of video columns | Q: Noise switch |
| E: Number of video rows | R: Bass Frequency |
| F: Character size: 0 = 8 × 8, 1 = 8 × 16 | S: Alto Frequency |
| G: Raster value | T: Soprano Frequency |
| H: Screen memory location | U: Noise Frequency |
| I: Character memory location | V: Loudness of sounds |
| J: Light pen—horizontal | W: Auxiliary color |
| K: Light pen—vertical | X: Screen color |
| L: Paddle 1 | Y: Reverse mode: 0 = on, 1 = off |
| M: Paddle 2 | Z: Border color |

C: Screen origin—vertical: This determines the up-down placement of the screen image. The normal value is 25. Lowering this causes the screen to move up by 2 rows of dots for each number lowered, and raising it moves the screen down.

To change value: POKE 36865, X

D: Number of video columns: Normally, this is set to 22.

Changing this will change the display accordingly. Numbers over 27 will give a 27 column screen. The cursor controls are based on a fixed number of 22 columns, and changing this number makes the cursor controls misbehave.

To change: POKE 36866, PEEK(36866) AND 128 OR X.

E: Number of video rows: The number of rows may range from 0 to 23. A larger number of rows causes garbage to appear on the bottom of the screen.

To change: POKE 36867, PEEK(36867) AND 129 OR (X*2)

F: Character size: This bit determined the size of the matrix used for each character. A 0 here sets normal mode, in which characters are 8 by 8 dots. A 1 sets 8 by 16 mode, where each character is now twice as tall. 8 by 16 mode is normally used for high resolution graphics, where it is likely to have many unique characters on the screen.

To set 8 by 8 mode: POKE 36867, PEEK(36867) AND 254

To set 8 by 16 mode: POKE 36867, PEEK(36867) OR 1

G: Raster value: This number is used to synchronize the light pen with the TV picture.

H: Screen memory location: This determines where in memory the VIC keeps the image of the screen. The highest bit in location 36869 must be a 1. Bits 4-6 of location 36869 are bits 10-12 of the screen's address, and bit 7 of location 36866 is bit 9 of the address of the screen. To determine the location of the screen, use the formula:

$$S = 4 * (\text{PEEK}(36866) \text{ AND } 128) + 64 * (\text{PEEK}(36869) \text{ AND } 112)$$

Note that bit 7 of location 36866 also determines the location of color memory. If this bit is a 0, color memory starts at location 37888. If this bit is a 1, color memory begins at 38400. Here is a formula for this:

$$C = 37888 + 4 * (\text{PEEK}(36866) \text{ AND } 128)$$

I: Character memory location: This determines where information on the shapes of characters are stored. Normally this pointer is to the character generator ROM, which contains both the upper case/graphics or the upper/lower case set. However, a simple POKE command can change this pointer to a RAM location, allowing custom character sets and high resolution graphics.

To change: POKE 36869, PEEK(36869) AND 15 OR(X*16)
(See chart on next page.)

J: Light pen horizontal: This contains the latched number of the dot under the light pen, from the left of the screen.

K: Light pen vertical: The latched number of the dot under the pen, counted from the top of the screen.

X Value	Location		Contents
	HEX	Decimal	
0	8000	32768	Upper case normal characters
1	8400	33792	Upper case reversed characters
2	8800	34816	Lower case normal characters
3	8C00	35840	Lower case reversed characters
4	9000	36864	unavailable
5	9400	37888	unavailable
6	9800	38912	VIC chip-unavailable
7	9400	39936	ROM-unavailable
8	0000	0	unavailable
9	—	—	unavailable
10	—	—	unavailable
11	—	—	unavailable
12	1000	4096	RAM
13	1400	5120	RAM
14	1800	6144	RAM
15	1C00	7168	RAM

L: Paddle X: This contains the digitized value of a variable resistance (game paddle). The number reads from 0 to 255.

M: Paddle Y: Same as Paddle X, for a second analog input.

N: Bass switch: If this bit is a 0, no sound is played from Voice 1. A 1 in this bit results in a tone determined by Frequency 1.

To turn on: POKE 36874, PEEK(36874) OR 128

To turn off: POKE 36874, PEEK(36874) AND 127

O: Alto switch: See Bass switch.

To turn on: POKE 36875, PEEK(36875) OR 128

To turn off: POKE 36875, PEEK(36875) AND 127

P: Soprano switch: See Bass switch.

To turn on: POKE 36876, PEEK(36876) OR 128

To turn off: POKE 36876, PEEK(36876) AND 127

Q: Noise switch: See Bass switch.

To turn on: POKE 36877, PEEK(36877) OR 128

To turn off: POKE 36877, PEEK(36877) AND 127

R: Bass Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

The actual frequency of the sound in cycles per second (hertz) is determined by the following formula:

$$\text{Frequency} = \frac{\text{Clock}}{(127 - X)}$$

X is the number from 0 to 127 that is put into the frequency register. If X is 127, then use - 1 for X in the formula. The value of Clock comes from the following table:

Register	NTSC (US TV's)	PAL (European)
36874	3995	4329
36875	7990	8659
36876	15980	17320
36877	31960	34640

To set: POKE 36874, PEEK(36874) AND 128 OR X

S: Alto Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

T: Soprano Frequency: This is a value corresponding to the frequency of the tone being played. The larger the number, the higher the pitch of the tone.

To set: POKE 36876, PEEK(36876) AND 128 OR X

U: Noise Frequency: This is a value corresponding to the frequency of the noise being played. The larger the number, the higher the pitch of the noise.

To set: POKE 36877, PEEK(36877) AND 128 OR X

V: Loudness of sounds: This is the volume control for all the sounds playing. 0 is off, and 15 is the loudest sound.

To set: POKE 36878, PEEK(36878) AND 240 OR X

W: Auxiliary color: This register holds the color number of the auxiliary color. The value can be from 0 to 15.

To set: POKE 36878, PEEK(36878) AND 15 OR (16*X)

X: Screen color: A number from 0 to 15 sets the color of the screen.

To set: POKE 36879, PEEK(36879) AND 240 OR X

Y: Reverse mode: A 1 in this bit indicates normal characters, and a 0 here causes all characters to be displayed as if reversed.

To turn on reverse mode: POKE 36879, PEEK(36879) AND 247

To turn off reverse mode: POKE 36879, PEEK(36879) OR 8

Z: Border color: A number from 0 to 7 sets the color of the screen.

To set: POKE 36879, PEEK(36879) AND 248 OR X

6522 VERSATILE INTERFACE ADAPTER

The 6522 Versatile Interface Adapter (VIA) provides the VIC with two peripheral ports with input latching, two powerful interval timers, and a serial-to-parallel/parallel-to-serial shift register.

Basically, the VIC chip handles Audio/Video input/output, and the 6522 handles the rest. . .cassette operations, joysticks, RS-232, and user port.

6522 Versatile Interface Adapter Description

ADDRESS	DESCRIPTION	REGISTER
9110	Port B	AAAAAAAA
9111	Port A (with handshaking)	BBBBBBBB
9112	Data Direction B	CCCCCCCC
9113	Data Direction A	DDDDDDDD
9114	Timer #1, low byte	EEEEEEEE
9115	Timer #1, high byte	FFFFFFF
9116	Timer #1, low byte to load	GGGGGGGG
9117	Timer #1, high byte to load	HHHHHHHH
9118	Timer #2, low byte	IIIIIIII
9119	Timer #2, high byte	JJJJJJJJ
911A	Shift Register	KKKKKKKK
911B	Auxiliary Control	LLMNNNOP
911C	Peripheral Control	QQRRSSST
911D	Interrupt Flags	UVWXYZab
911E	Interrupt Enable	cdefghij
911F	Port A (no handshaking)	kkkkkkkk

PORT A I/O REGISTER

These eight bits are connected to the eight pins which make up port B. Each pin can be set for either input or output.

Input latching is available on this port. When latch mode is enabled the data in the register freezes when the CB1 interrupt flag is set. The register stays latched until the interrupt flag is cleared.

Handshaking is available for output from this port. CB2 will act as a DATA READY SIGNAL. This must be controlled by the user program. CB1 acts as the DATA ACCEPTED signal, and must be controlled by the device connected to the port. When DATA ACCEPTED is sent to the 6522, the DATA READY line is cleared, and the interrupt flag is set.

PORT B I/O REGISTER

These eight bits are connected to the eight pins which make up port A. Each pin can be set for either input or output. Handshaking is available for both read and write operations. Write handshaking is similar to that on PORT B. Read handshaking is automatic. The CA1 input pin acts as a DATA READY signal. The CA2 pin (used for output) is used for a DATA ACCEPTED signal. When a DATA READY signal is received a flag is set. The chip can be set to generate an interrupt or the flag can be polled under program control. The DATA ACCEPTED signal can either be a pulse or a DC level. It is set low by the CPU and cleared by the DATA READY signal.

DATA DIRECTION FOR PORT B

This register is used to control whether a particular bit in PORT B is used for input or output. Each bit of the data direction register (DDR) is associated with a bit of port B. If a bit in the DDR is set to 1, the corresponding bit of the port will be an OUTPUT. If a bit in the DDR is 0, the corresponding bit of the port will be an INPUT.

For example, if the DDR is set to 7, port B will be set up as follows:

BITS	NUMBER	DDR	PORT B FUNCTION
	0	1	OUTPUT
	1	1	OUTPUT
	2	1	OUTPUT
	3	0	INPUT
	4	0	INPUT
	5	0	INPUT
	6	0	INPUT
	7	0	INPUT

DATA DIRECTION REGISTER FOR PORT A

This is similar to the DDR for port B, except that it works on PORT A.

E,F,G,H: TIMER CONTROLS

There are two timers on the 6522 chip. The timers can be set to count down automatically or count pulses received by the VIA. The mode of operation is selected by the Auxiliary Control register.

TIMER T1 on the 6522 consists of two 8-bit latches and a 16-bit counter. The various modes of the TIMER are selected by setting the AUXILIARY CONTROL REGISTER (ACR). The latches are

used to store a 16-bit data word to load into the counter. Loading a number into the latches does not affect the count in progress.

After it is set, the counter will begin decrementing at 1MHz. When the counter reaches zero, an interrupt flag will be set, and the IRQ will go low. Depending on how the TIMER is set, either further interrupts will be disabled, or it will automatically load the two latches into the counter and continue counting. The TIMER can also be set to invert the output signal on a peripheral pin each time it reaches zero and resets.

The TIMER locations work differently on reading and writing.

WRITING TO THE TIMER:

E: Write into the low order latch. This latch can be loaded into the low byte of the 16-bit counter.

F: Write into the high order latch, write into the high order counter, transfer low order latch into the low order counter, and reset the TIMER T1 interrupt flag. In other words, when this location is set the counter is loaded.

G: Same as E.

H: Write into the high order latch and reset the TIMER T1 interrupt flag.

READ TIMER T1

E: Read the TIMER T1 low order counter and reset the TIMER T1 interrupt flag.

F: Read the TIMER T1 high order counter.

G: Read the TIMER T1 low order latch.

H: Read the TIMER T1 high order latch.

TIMER T2

This TIMER operates as an interval timer (in one-shot mode), or as a counter for counting negative pulses on PORT B pin 6. A bit in the ACR selects which mode TIMER T2 is in.

WRITING TO TIMER T2

I: Write TIMER T2 low order byte of latch.

J: Write TIMER T2 high order counter byte, transfer low order latch to low order counter, clear TIMER T2 interrupt flag.

READING TIMER T2

I: Read TIMER T2 low order counter byte, and clear TIMER T2 interrupt flag.

J: Read TIMER T2 high order counter byte.

K: SHIFT REGISTER

A shift register is a register which will rotate itself through the CB2 pin. The shift register can be loaded with any 8-bit pattern which can be shifted out through the CB1 pin, or input to the CB1 pin can be shifted into the shift register and then read. This makes it highly useful for serial to parallel and parallel to serial conversions.

The shift register is controlled by bits 2-4 of the Auxiliary Control register.

L,M,N,O,P: AUXILIARY CONTROL REGISTER

L: TIMER 1 CONTROL

BIT #	7	6	
	0	0	One-shot mode (output to PB7 disabled)
	0	1	Free running mode (output to PB7 disabled)
	1	0	One-shot mode (output to PB7 enabled)
	1	1	Free running mode (output to PB7 enabled)

M: TIMER 2 CONTROL

TIMER 2 has 2 modes. If this bit is 0, TIMER 2 acts as an interval timer in one-shot mode. If this bit is 1, TIMER 2 will count a predetermined number of pulses on pin PB6.

N: SHIFT REGISTER CONTROL

BIT #	4	3	2	
	0	0	0	SHIFT REGISTER DISABLED
	0	0	1	SHIFT IN (FROM CB1) UNDER CONTROL OF TIMER 2
	0	1	0	SHIFT IN UNDER CONTROL OF SYSTEM CLOCK PULSES
	0	1	1	SHIFT IN UNDER CONTROL OF EXTERNAL CLOCK PULSES
	1	0	0	FREE RUN MODE AT RATE SET BY TIMER 2
	1	0	1	SHIFT OUT UNDER CONTROL OF TIMER 2
	1	1	0	SHIFT OUT UNDER CONTROL OF SYSTEM CLOCK PULSES
	1	1	1	SHIFT OUT UNDER CONTROL OF EXTERNAL CLOCK PULSES

O: PORT B LATCH ENABLE

As long as this bit is 0, the PORT B register will directly reflect the data on the pins.

If this bit is set to one, the data present on the input pins of PORT A will be latched within the chip when the CB1 INTERRUPT FLAG is set. As long as the CB1 INTERRUPT FLAG is set, the data on the pins can change without affecting the contents of the PORT B register. Note that the CPU always reads the register (the latches) rather than the pins.

Input latching can be used with any of the input or output modes available for CB2.

P: PORT A LATCH ENABLE

As long as this bit is 0, the PORT A register will directly reflect the data on the pins.

If this bit is set to one, the data present on the input pins of PORT A will be latched within the chip when the CA1 INTERRUPT FLAG is set. As long as the CA1 INTERRUPT FLAG is set, the data on the pins can change without affecting the contents of the PORT A register. Note that the CPU always reads the register (the latches) rather than the pins.

Input latching can be used with any of the input or output modes available for CA2.

Q,R,S,T THE PERIPHERAL CONTROL REGISTER

Q: CB2 CONTROL

	Q	Q	Q	
BIT #	7	6	5	DESCRIPTION
	0	0	0	Interrupt Input Mode
	0	0	1	Independent Interrupt Input Mode
	0	1	0	Input Mode
	0	1	1	Independent Input Mode
	1	0	0	Handshake Output Mode
	1	0	1	Pulse Output Mode
	1	1	0	Manual Output Mode (CB2 is held LOW)
	1	1	1	Manual Output Mode (CB2 is held HIGH)

INTERRUPT INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a negative (high-to-low) transition on the CB2 input line. The CB2 interrupt bit will be cleared on a read or write to PORT B.

INDEPENDENT INTERRUPT INPUT MODE:

As above, the CB2 interrupt flag will be set on a negative transition on the CB2 input line. However, reading or writing to PORT B does not clear the flag.

INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a positive (low-to-high) transition of the CB2 line. The CB2 flag will be cleared on a read or write of PORT B.

INDEPENDENT INPUT MODE:

As above, the CB2 interrupt flag will be set on a positive transition on the CB2 line. However, reading or writing PORT B does not affect the flag.

HANDSHAKE OUTPUT MODE:

The CB2 line will be set low on a write to PORT B. It will be reset high again when there is an active transition on the CB1 line.

PULSE OUTPUT MODE:

The CB2 line is set low for one cycle after a write to PORT B.

MANUAL OUTPUT MODE:

The CB2 line is held low.

MANUAL OUTPUT MODE:

The CB2 line is held high.

R: CB1 CONTROL

This bit selects the active transition of the input signal applied to the CB1 pin. If this bit is 0, the CB1 interrupt flag will be set on a negative transition (high-to-low). If this bit is a 1, the CB1 interrupt flag will be set on a positive (low-to-high) transition.

S: CA2 CONTROL

	S	S	S	
BIT #	3	2	1	DESCRIPTION
	0	0	0	Interrupt Input Mode
	0	0	1	Independent Interrupt Input Mode
	0	1	0	Input Mode
	0	1	1	Independent Input Mode

1	0	0	Handshake Output Mode	
1	0	1	Pulse Output Mode	
1	1	0	Manual Output Mode	(CA2 is held LOW)
1	1	1	Manual Output Mode	(CA2 is held HIGH)

INTERRUPT INPUT MODE:

The CA2 interrupt flag (IFR bit 0) will be set on a negative (high-to-low) transition on the CA2 input line. The CA2 interrupt bit will be cleared on a read or write to PORT A.

INDEPENDENT INTERRUPT INPUT MODE:

As above, the CA2 interrupt flag will be set on a negative transition on the CA2 input line. However, reading or writing to PORT A does not clear the flag.

INPUT MODE:

The CA2 interrupt flag (IFR bit 0) will be set on a positive (low-to-high) transition of the CA2 line. The CA2 flag will be cleared on a read or write of PORT A.

INDEPENDENT INPUT MODE:

As above, the CA2 interrupt flag will be set on a positive transition on the CA2 line. However, reading or writing PORT A does not affect the flag.

HANDSHAKE OUTPUT MODE:

The CA2 line will be set low on a read or write to PORT A. It will be reset high again when there is an active transition on the CA1 line.

PULSE OUTPUT MODE:

The CA2 line is set low for one cycle after a read or write to PORT A.

MANUAL OUTPUT MODE:

The CA2 line is held low.

MANUAL OUTPUT MODE:

The CA2 line is held high.

T: CA1 CONTROL

This bit of the PCR selects the active transition of the input signal applied to the CA1 input pin. If this bit is 0, the CA1 interrupt flag (Bit) will be set by a negative transition (high-to-low) on the CA1 pin. If this bit is 1, the CA1 interrupt flag will be set by a positive transition (low-to-high).

There are two registers associated with interrupts: The INTERRUPT FLAG REGISTER (IFR) and the INTERRUPT ENABLE REGISTER (IER). The IFR has eight bits, each one connected to a register in the 6522. Each bit in the IFR has an associated bit in the IER. The flag is set when a register wants to interrupt. However, no interrupt will take place unless the corresponding bit in the IER is set.

UVWXYZab: INTERRUPT FLAG REGISTER

When the flag is set, the pin associated with that flag is attempting to interrupt the 6502. Bit U is not a normal flag. It goes high if both the flag and the corresponding bit in the INTERRUPT ENABLE REGISTER are set. It can be cleared only by clearing all the flags in the IFR or disabling all active interrupts in the IER.

	SET BY	CLEARED BY
U	IRQ STATUS	
V	TIMER 1 time-out	Reading TIMER 1 low order counter and writing TIMER 1 high order latch
W	TIMER 2 time-out	Reading TIMER 2 low order counter and writing TIMER 2 high order counter
X	CB1 pin active transition	Reading or writing PORT B
Y	CB2 pin active transition	Reading or writing PORT B
Z	Completion of 8 shifts	Reading or writing the shift register
a	CA1 pin active transition	Reading or writing PORT A (BBBBBBBBB in above chart)
b	CA2 pin active transition	Reading or writing PORT A (BBBBBBBBB in above chart)

cdefghij: INTERRUPT ENABLE REGISTER

c: ENABLE CONTROL

If this bit is a 0 during a write to this register, each 1 in bits 0-6

clears the corresponding bit in the IER. If this bit is a 1 during this register, each 1 in bits 0-6 will set the corresponding IER bit.

- d** **TIMER 1 time-out enable**
- e** **TIMER 2 time-out enable**
- f** **CB1 interrupt enable**
- g** **CB2 interrupt enable**
- h** **Shift interrupt enable**
- i** **CA1 interrupt enable**
- j** **CA2 interrupt enable**

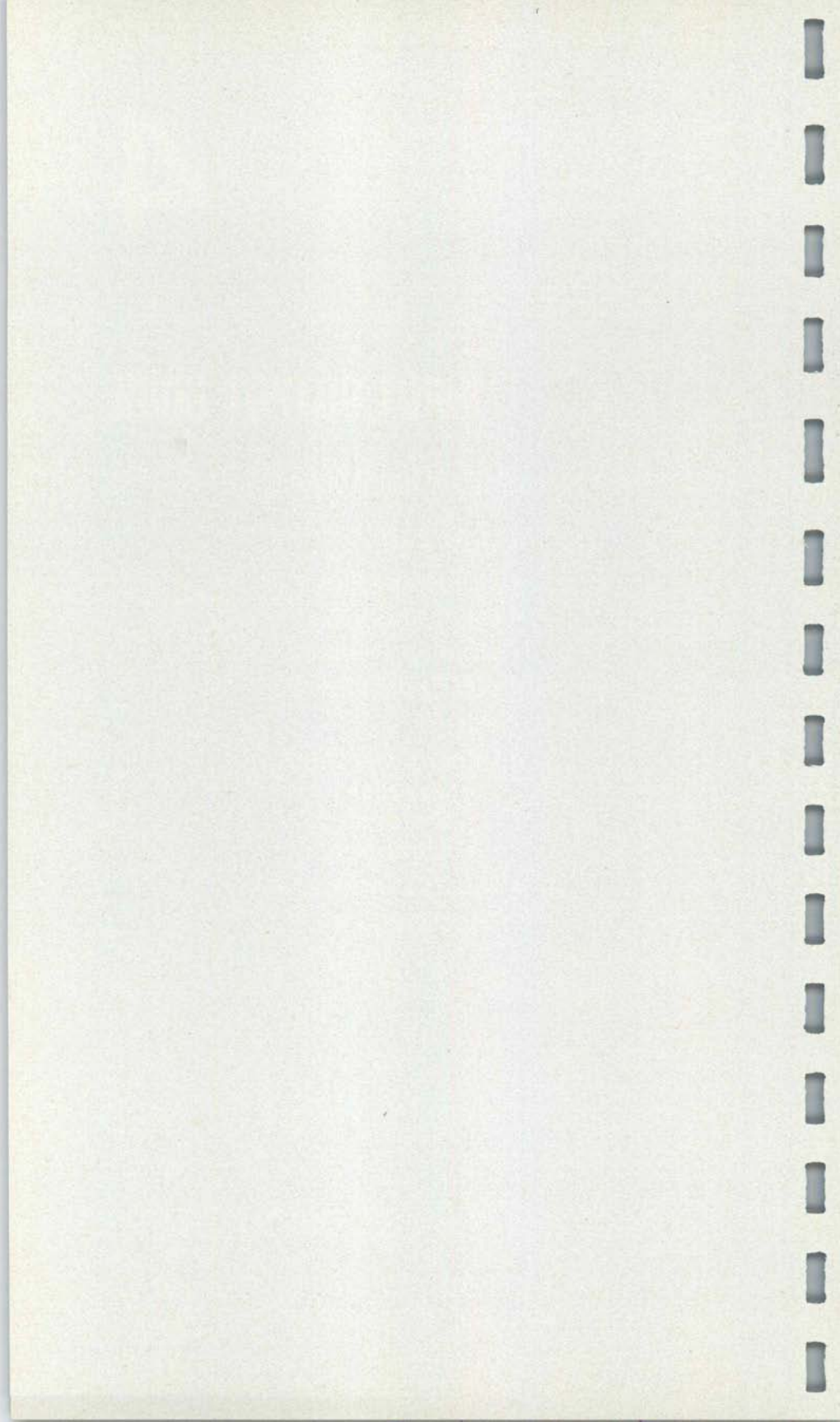
kkkkkkkk: PORT A

This is similar to BBBBBBBB, except that the handshaking lines (CA1 and CA2) are unaffected by operations on this port.

4

INPUT/OUTPUT GUIDE

- User Port
- The Serial Bus
- Using the VIC Graphic Printer
- VIC Expansion Port
- Game Controllers
 - Joystick
 - Paddles
 - Light Pen
- RS-232 Interface Description



THE USER PORT

The user port is meant to connect the VIC to the outside world. By using the lines available at this port, you can connect the VIC to a printer, a Votrax Type and Talk, a MODEM, a second joystick, even another computer.

The port on the VIC is directly connected to one of the 6522 VIA chips. By programming, the VIA will connect to many other devices.

Port Pin Description

PIN #	DESCRIPTION
UPPER SIDE	
1	GROUND
2	+5V (100mA MAX.)
3	RESET By grounding this pin, the VIC will do a COLD START, resetting completely and erasing any program in memory.
4	JOY 0 This pin is connected to joystick switch 0 (See GAME PORT).
5	JOY 1 (See GAME PORT.)
6	JOY 2 (See GAME PORT.)
7	LIGHT PEN This pin also acts as the input for the joystick FIRE button (See GAME PORT).
8	CASSETTE SWITCH This pin is connected to the SENSE cassette switch line.
9	SERIAL ATN IN This pin is connected to the ATN IN line of the serial bus.
10	9VAC Connected directly to the VIC transformer
11 &	9VAC
12	GND
BOTTOM SIDE	
A	GND
B	CB1
C	PB0
D	PB1
E	PB2
F	PB3
H	PB4
J	PB5
K	PB6

L	PB7	be set up as either an INPUT or an OUTPUT
M	CB2	by setting the DATA DIRECTION REGISTER properly. It is located at 37138 (\$9112 hex).
N	GND	

Each of the eight lines in the PORT has a BIT in the 8 bit DATA DIRECTION REGISTER (DDR) which controls whether that line will be an input or an output. If a bit in the DDR is a ONE, the corresponding line of the PORT will be an OUTPUT. If a bit in the DDR is a ZERO, the corresponding line of the PORT will be an INPUT. For example, if bit 3 of the DDR is set to 1, then line 3 of the PORT will be an output. As another example, if the DDR is set like this:

BIT #:	7 6 5 4 3 2 1 0
VALUE:	0 0 1 1 1 0 0 0

You can see that lines 5,4, and 3 will be outputs since those bits are ones. The rest of the lines will be inputs, since those lines are zeros.

To PEEK or POKE the USER port, it is necessary to use both the DDR and the PORT itself.

Remember that the PEEK and POKE statements want a number from 0-255. The numbers given in the example must be translated into decimal before they could be used. (The value would be: $2^5 + 2^4 + 2^3 = 32 + 16 + 8 = 56$. See Section 1 on numbers for more details.)

The other two lines, CB1 and CB2 are different from the rest of the USER PORT. These two lines are mainly for HANDSHAKING, and are programmed differently from port B.

Handshaking is needed when two devices communicate. Since one device may run at a different speed than another device, it is necessary to give the devices some way of knowing what the other is doing. Even when the devices are operating at the same speed, handshaking is necessary to let the other know when data is to be sent, and if it has been received. Both the CB1 and CB2 lines have special characteristics which make them well suited for handshaking.

CB1 is usually used as an input (except under SHIFT REGISTER control). CB2 can be used both for input and output, but is usually used for output.

It is not possible to read CB1 directly. CB1 is designed to set a flag (bit 4) in the INTERRUPT FLAG register (located at 37149 or \$911D HEX) when a transition occurs on the CB1 line. Bit 4 of the PERIPHERAL CONTROL REGISTER (PCR) located at 37148 (\$911C hex) determines whether CB1 will set the flag on a

low-to-high transition or on a high-to-low transition. Once the CB1 flag is set, it will stay set until you clear it by a Peek or POKE to PORT B which resets the CB1 flag bit. If bit 4 in the INTERRUPT ENABLE register is set, and interrupts are enabled, the transition will also cause an INTERRUPT REQUEST (IRQ).

CB2 is controlled by the PCR. Bits 7,6 control whether CB2 will be an input or an output. Bit 5 controls the setting of CB2.

BIT #	7	6	5	DESCRIPTION
	0	0	0	Interrupt Input Mode
	0	0	1	Independent Interrupt Input Mode
	0	1	0	Input Mode
	0	1	1	Independent Input Mode
	1	0	0	Handshake Output Mode
	1	0	1	Pulse Output Mode
	1	1	0	Manual Output Mode (CB2 is held LOW)
	1	1	1	Manual Output Mode (CB2 is held HIGH)

INTERRUPT INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a negative (high-to-low) transition on the CB2 input line. The CB2 interrupt bit will be cleared on a read or write to PORT B.

INDEPENDENT INTERRUPT INPUT MODE:

As above, the CB2 interrupt flag will be set on a negative transition on the CB2 input line. However, reading or writing to PORT B does not clear the flag.

INPUT MODE:

The CB2 interrupt flag (IFR bit 3) will be set on a positive (low-to-high) transition of the CB2 line. The CB2 flag will be cleared on a read or write of PORT B.

INDEPENDENT INPUT MODE:

As above, the CB2 interrupt flag will be set on a positive transition on the CB2 line. However, reading or writing PORT B does not affect the flag.

HANDSHAKE OUTPUT MODE:

The CB2 line will be set low on a write to PORT B. It will be reset high again when there is an active transition on the CB1 line.

PULSE OUTPUT MODE:

The CB2 line is set low for one cycle after a write to PORT B.

MANUAL OUTPUT MODE:

The CB2 line is held low.

MANUAL OUTPUT MODE:

The CB2 line is held high.

MORE MUSIC FOR THE VIC

Now that you know about the USER PORT, there is little surprise left. Up to now, the VIC has had 4 musical voices . . . three music registers and a white noise register. By connecting a small amplifier and speaker to the USER PORT, and doing a little programming, you can get another musical voice.

THEORY

Most music is made up of square waves of different amplitudes and frequencies. One of the functions of the 6522 chip is to generate square waves through the CB2 line. If we connect the CB2 line to a speaker, we will be able to hear the square waves generated by the VIC.

NOTE: Connecting a speaker directly to CB2 may damage your VIC. You must connect the speaker through an amplifier to protect the VIC.

PARTS NEEDED

1. Small battery powered speaker/amplifier
2. User Port Connector (12 position, 24 contact edge connector with .156" spacing)
3. Wire

CONNECTING TO YOUR VIC

1. Wire the GROUND of the amplifier to the GROUND of the USER PORT (pin N).
2. Wire the SIGNAL of the amplifier to the CB2 output of the USER PORT (pin M).

You are now ready to add your other voice through a BASIC program.

BASIC PROGRAM STEPS:

1. Set the 6522 shift register to free running mode by:

POKE 37147,16

2. Set the shift rate by:

POKE 37144,C where C is an integer from 0 to 255
C is the note to be played.

3. Load the shift register by:

POKE 37146,D where D = 15, 51, or 85 for a square wave.
This step sets the octave for the note.

This step must be done last, since as soon as it is set, the VIC starts generating the square waves.

The frequency of the square wave can be found by the following formula:

$$\text{FREQUENCY} = \frac{500000 \text{ Hz}}{(C+2) (D1)} \quad \begin{array}{l} \text{Where } D1=8 \text{ when } D=15 \\ D1=4 \text{ when } D=51 \\ D1=2 \text{ when } D=85 \end{array}$$

When you are in this mode, the VIC will not read or write to cassette. To restore normal operations, you must:

POKE 37147,0

The following program demonstrates music using this method. By hitting a letter the note will be played.

```
10 PRINT "MUSICAL USING CB2."
15 PRINT "HIT + TO GO UP AN OCTAVE"
16 PRINT "HIT - TO GO DOWN AN OCTAVE"
17 PRINT: PRINT "USE E TO EXIT."
20 POKE37147,16:DIMA(30):FOR K=1 TO 30:READ A(K):
    NEXT
40 GET A$:IF A$="" THEN 40
42 IF A$="E" THEN POKE37147,0:END
45 IF A$="+" THEN SF=SF-(SF<2):GOTO 40
50 IF A$="-" THEN SF=SF+(SF<0):GOTO 40
60 A=8-ASC(A$)+64:IF A>7 OR A<1 THEN 40
70 POKE 37144,A(A-(SF=1)*10-(SF=2)*20)
80 POKE37146,-(SF=0)*15-(SF=1)*51-(SF=2)*85
90 GOTO 40
100 DATA 59,61,65,69,73,77,82,87,90,93
110 DATA 99,104,111,117,120,124,132,140,149,157
120 DATA 167,177,182,188,199,211,224,237,244,251
```

THE SERIAL BUS

The serial bus is a daisy chain arrangement designed to let the VIC communicate with the VIC-1540 DISK DRIVE and the VIC-1515 GRAPHICS PRINTER. Up to 5 devices can be connected to the serial bus at one time.

All devices connected on the serial bus will receive all the data transmitted over the bus. To allow the VIC to route data to its intended destination, each device has a bus ADDRESS. By using this device address, the VIC can control access to the bus. Addresses on the serial bus range from 4 to 31.

The VIC can COMMAND a particular device to TALK or LISTEN. When the VIC commands a device to TALK, the device will begin putting data onto the serial bus. When VIC commands a device to LISTEN, the device addressed will get ready to receive data (from the VIC or from another device on the bus). Only one device can TALK on the bus at a time; otherwise the data will collide and the system will crash in confusion. However, any number of devices can LISTEN at the same time to one TALKER.

COMMON SERIAL BUS ADDRESSES

Number	Device
4 or 5	VIC-1515 GRAPHIC PRINTER
8	VIC DISK DRIVE

Other device addresses are possible. Each device is wired to an address. Certain devices (like the VIC printer) provide a choice between two addresses for the convenience of the user.

The SECONDARY ADDRESS is to let the VIC transmit set up information to a device. For example, to OPEN a connection on the bus to the printer, and have it print in UPPER/LOWER case, use the following:

OPEN 1,4,7

Where 1 is the logical file number (the number you PRINT# to)
4 is the ADDRESS of the printer

and 7 is the SECONDARY ADDRESS that tells the printer to go into UPPER/LOWER case mode.

SERIAL BUS PINOUTS

PIN #	
1	SERIAL SRQ IN
2	GND

3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	NO CONNECTION

SERIAL SRQ IN: (SERIAL SERVICE REQUEST IN)

Any device on the serial bus can bring this signal LOW when it requires attention from the VIC. The VIC will then take care of the device.

SERIAL ATN IN/OUT: (SERIAL ATTENTION IN/OUT)

The VIC uses this signal to start a command sequence for a device on the serial bus. When the VIC brings this signal LOW, all other devices on the bus start listening for the VIC to transmit an address. The device addressed must respond in a preset period of time; otherwise the VIC will assume that the device addressed is not on the bus, and will return an error in the STATUS WORD.

SERIAL CLK IN/OUT: (SERIAL CLOCK IN/OUT)

This signal is used for timing on the serial bus.

SERIAL DATA IN/OUT:

Data on the serial bus is transmitted one bit at a time on this line.

USING THE VIC GRAPHIC PRINTER

The VIC Graphic Printer connects to the serial port on the back of the VIC and is used for printing out listings of programs, statistical data, graphs, charts and even graphic plotting. The VIC printer can also be used with the VICWriter wordprocessing cartridge to write reports, transcribe school notes, write letters and other documents.

Programmers use dot matrix printers to make paper copies of program listings which are easier to debug and edit on paper. It's also helpful to print out text and graphics displayed on the screen through what is called a "screen dump to the printer."

Several short programs which demonstrate the use of the printer are included below. We've even included a special "typing" program which lets you use the VIC as a typewriter to enter words or graphics directly from the keyboard.

LISTING DATA ON YOUR VIC PRINTER

The proper format for printing out a listing of a program which resides in memory is to enter the following line and type RUN.

```
OPEN4,4:CMD4:LIST
```

Note that if you have left a printer file open you will get a FILE OPEN error. If this happens, type CLOSE4, and hit RETURN. Then retype the above.

If you're developing a new program, you probably want to list out each revision so you can edit it on paper before proceeding to the next step. A good way to do this is to (1) type in your program, (2) include an END statement at the end of the program, (3) include the list-to-printer instruction at a high line number above the END statement, and (4) RUN the line with the printer instruction whenever you want to list out the program. This technique lets you RUN and edit your program normally, but it saves you the trouble of having to type in the LIST line every time you want a listing on paper. In the following example, if you type RUN, the program will print out line 10. If you type RUN 5000, the program will list to the printer.

```
10 PRINT"THIS IS MY PROGRAM"  
20 END  
5000 CLOSE4: OPEN4,4:CMD4:LIST
```

VIC GRAPHIC PRINTER COMMAND CHART

The VIC Graphic Printer includes a strong "language" of special print commands, as described in the following chart. Insert the CHR\$ command to use these commands:

10 PRINT#4,CHR\$ (14) "PUT DATA HERE".

PRINT COMMAND	DESCRIPTION
CHR\$ (10)	Line feed after printing
CHR\$ (13)	Carriage return
CHR\$ (8)	Graphic mode command
CHR\$ (14)	Double width characters
CHR\$ (15)	Standard character mode (type this to get back to normal)
CHR\$ (16)	Print start position addressing
CHR\$ (27)	When followed by the CHR\$ (16) position code this command is used to specify a start position by dot address (in contrast to character address)
CHR\$ (26)	Repeat graphic select command
CHR\$ (145)	Cursor up (Upper case) mode
CHR\$ (17)	Cursor down (Upper/Lower case) mode
CHR\$ (18)	Reverse field on command
CHR\$ (146)	Reverse field off command

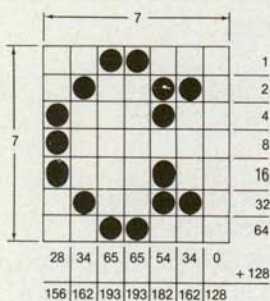
PRINTING DOT PROGRAMMABLE GRAPHICS

There are two fairly easy ways to print your own graphic characters—in other words, "dot programmable graphics." One way is to purchase Commodore's PROGRAMMABLE CHARACTER SET & GAMEGRAPHICS EDITOR, which is available on tape at an economical price. The other way is to define your own characters TO THE PRINTER using a character matrix of 7×7 dots. You have to define each character in terms of its binary code and the best way to do this is to use DATA statements. Look at the following matrix and decide which dot pattern you would like to design. A sample is shown. Now, to print this out, you have to add up the binary values for each column. Count zero for empty blocks, count the value shown on the left of the pattern if there is a dot there, then add up all the values for the column. The total of each column is shown at the bottom in our sample on page 238.

PRINTING IN DIRECT MODE

You can use your printer like a typewriter by printing in the DIRECT MODE. In this mode, the printer prints everything between the quotation marks including graphics and reversed characters. Here's a sample of how it's done:

SAMPLE PROGRAM



```

10 DATA156,162,193,193,182,162
20 FORI=1TO6
30 READA
40 A$=A$+CHR$(A)
50 NEXT
60 OPEN4,4
70 FORI=1TO4
80 PRINT#4,CHR$(8)A$;
90 PRINT#4,CHR$(15)" COMMODORE"
100 NEXT
    
```

After typing RUN, you get this result:

```

Q COMMODORE
Q COMMODORE
Q COMMODORE
Q COMMODORE
    
```

YOU TYPE:

SCREEN DISPLAYS: PRINTER PRINTS:

OPEN4,4	OPEN4,4 READY.	No response.
CMD4	No Response.	READY.
PRINT#4, "HELLO, LOVE "	PRINT#4, "HELLO, LOVE "	HELLO,LOVE
CLOSE4	CLOSE4 READY.	READY.

PRINTING DOUBLE WIDTH CHARACTERS

Double width characters have many applications, from enlarging graphics on paper to printing bold face headlines and titles. The following program demonstrates how to use CHR\$ (14) to print double spaced letters . . . and also shows how to get back to normal letters by typing CHR\$ (15).

```

10 OPEN4,4:PRINT#4,CHR$ (14) "DOUBLE LETTERS"
20 PRINT#4, "STILL DOUBLE"
    
```

```
30 PRINT#4, CHR$(15) "NORMAL AGAIN"
40 PRINT"HELLO AGAIN IN NORMAL MODE"
```

(If you change PRINT#4 to CMD4 in line 30 it still works.)

PRINTING REVERSE CHARACTERS

You may use the codes shown in the VIC Graphic Printer Command chart to tell the printer to print reverse characters by including these lines:

```
Reverse On: 10 OPEN4,4: PRINT#4,CHR$(18)
Print Info: 20 PRINT#4, "VIC 20"
Reverse Off: 30 PRINT#4,CHR$(146)
Normal Again: 40 PRINT#4,"VIC 20"
```

PRINTING WHAT IS DISPLAYED ON THE SCREEN

Type and run these lines as a program or as a subroutine to get a printout of what is displayed on the screen from your program. When you use it, add a line or command that says: GOSUB 60000 and enter this program as shown. *Lines 10 through 25 are included to give you an example of how this can work and are not part of the screen dump program.* Your own program would have different commands here, of course.

You can type different "screens" of information in the course of a program. One way to do this is to add a line in your program that would scan the keyboard and look for a function key. Try adding this line as the first line of your program: 10 GETX\$:IFX\$= CHR\$(133) THEN GOSUB60000. This line lets you print out whatever is on the screen whenever you hit Function Key 1.

```
10 PRINT "SAMPLE"
20 GOSUB60000
25 END
60000 REM SCREEN COPY
60010 R$= CHR$(145):V$= CHR$(146):OPEN4,
4:PRINT#4:G= PEEK(648)*256:PRINT#4,R$FORP=
GTOG+505
60020 C=PEEK(P):C$="":IF(P-G)/22=INT((P-G)/22)THEN
PRINT#4,CHR$(8)+CHR$(13)+CHR$(15);
60030 IFC>128THENC=C-128:C$= CHR$(18)
60040 IFC<320RC>95THENC=C+64:GOTO60060
60050 IFC>63ANDC<96THENC=C+128
60060 C$= C$+ CHR$(C):IFLEN(C$)>1THENC$= C$+
V$+ R$
60070 PRINT#4,C$; :NEXT:PRINT#4:CLOSE4:RETURN
```

This port is used for expansions of the VIC system which require access to the address bus or the data bus of the computer. Caution is necessary when using the expansion bus, as it is possible to damage the VIC by malfunctioning user equipment.

The signals available on the connector are as follows:

NAME	PIN #	DESCRIPTION
GND	1	System ground
CD0	2	Data bus bit 0
CD1	3	Data bus bit 1
CD2	4	Data bus bit 2
CD3	5	Data bus bit 3
CD4	6	Data bus bit 4
CD5	7	Data bus bit 5
CD6	8	Data bus bit 6
CD7	9	Data bus bit 7
BLK1	10	8K decoded RAM/ROM block 1, @ \$2000 (active low)
BLK2	11	8K decoded RAM/ROM block 2, @ \$4000 (active low)
BLK3	12	8K decoded RAM/ROM block 3, @ \$6000 (active low)
BLK5	13	8K decoded ROM block 5, @ \$A000 (active low)
RAM1	14	1K decoded RAM block, @ \$0400 (active low)
RAM2	15	1K decoded RAM block @ \$0800 (active low)
RAM3	16	1K decoded RAM block @ \$0C00 (active low)
VR/W	17	Read/Write line from VIC chip (high-read, low-write)
CR/W	18	Read/Write line from CPU (high-read, low-write)
IRQ (NC)	19 20	Interrupt Request line to 6502 (active low)
+5V	21	
GND	22	
GND	A	
CA0	B	Address bus bit 0
CA1	C	Address bus bit 1
CA2	D	Address bus bit 2
CA3	E	Address bus bit 3
CA4	F	Address bus bit 4

CA5	H	Address bus bit 5
CA6	J	Address bus bit 6
CA7	K	Address bus bit 7
CA8	L	Address bus bit 8
CA9	M	Address bus bit 9
CA10	N	Address bus bit 10
CA11	P	Address bus bit 11
CA12	R	Address bus bit 12
CA13	S	Address bus bit 13
<u>I/O2</u>	T	I/O block 2 (located at \$9600)
<u>I/O3</u>	U	I/O block 3 (located at \$9C00)
<u>S/O2</u>	V	Phase 2 system clock
<u>NMI</u>	W	6502 Non Maskable Interrupt (active low)
<u>RESET</u>	X	6502 RESET pin (active low)
(NC)	Y	
GND	Z	

RAM Signals—RAM 1, 2, and 3 are active low signals which are used to decode memory placed in the 3K block from \$0400 to \$1000. Each of the RAM signals controls a 1K block of memory. When a RAM signal goes low, it indicates that the block of memory it controls is being addressed.

BLK Signals—The four block signals are also for memory expansion of the system. In this case, however, each decodes a different 8K block of memory. As with the RAM signals, each is active low. Blocks 1, 2, and 3 can be used for either RAM or ROM. Memory in those locations can (and will) be used by BASIC. Memory in Block 5, however, should be ROM, as this area is not accessible to BASIC. If RAM is placed here it can only be utilized by machine language programs.

IMPORTANT NOTE: If the additional memory is added to the VIC using the BLOCK decoding signals, memory added by using the RAM signals can not be used by BASIC for storage of BASIC text. This is because BASIC demands a continuous area for programs. With additional memory in the BLOCK decoded areas, the screen moves to \$1000, breaking up the area. Memory placed in those RAM areas can still be used by machine language programs.

IRQ—This is the interrupt request line. The VIC uses this internally for keyboard scan and the system clock.

RESET—When this line is grounded, it causes a COLD START of the VIC. Everything is RESET, including memory, so any program in the VIC at that time is destroyed.

NMI—When this line is grounded, it causes a VIC WARM START (just like RUN/STOP-RESTORE).

Address Bus—The address bus controls what memory location or I/O device the VIC will read from or write to. Only 14 bits appear on the connector, even though the address bus size is 16 bits, because two bits are decoded into the BLOCK and I/O signals.

Data Bus—The data bus is used by the VIC to move data to or from memory or I/O devices.

I/O Signals—These two signals can be used to add additional I/O devices to the VIC. The IEEE adapter from Commodore uses these signals.

Read/Write Signals—These signals inform the memory or the device being addressed whether the VIC wants to write data or read data. If the signal is high, a read is expected. If the signal is low, a write is desired.

There are two R/W signals available on the expansion port. One (CR/W) is connected to the 6502. The other (VR/W) is connected to the VIC chip. Memory expansion will normally use the VR/W signal. Other devices may need the CR/W signal.

WHAT HAPPENS WHEN MEMORY IS EXPANDED

The VIC comes with 5K of random access memory (RAM) located from 0 to 1023 (\$0000 to \$03FF) for operating system use, 4096-8192 (\$1000 to \$1FFF) which is BASIC program area, and from 38400 to 38911 (\$9600 to \$97FF) which is COLOR memory area.

When additional memory is added the VIC screen location, color memory location, or the start of BASIC might change.

Start of Added Memory	Start of Screen	Start of Color Memory	Start of BASIC
1024-4095	7680 (\$1E00)	38400 (\$9600)	1024 (\$0400)
8192 on up (\$2000-3FFF) (\$4000-5FFF) (\$6000-7FFF)	4096 (\$1000)	37888 (\$9400)	4608 (\$1200)

The VIC has 2 areas to add additional memory—a 3K space from 1024 to 4095 (\$0400 to \$0FFF) and a 24K section from 8192 to

32767 (\$2000 to \$7FFF). When the large expansion area is used, BASIC cannot use the 3K area as program area.

When memory is added in the 3K area, the BASIC program area will start at the beginning of the new RAM area. The screen will still begin at 7680 and color memory will still begin at 38400. However, the start of BASIC will be at 1024.

The VIC chip cannot access any of this new memory, so screen memory and programmable character memory must be in VIC internal memory (4096 to 8191).

Memory is added to the larger expansion area in 8K blocks, beginning at 8192 (\$2000). BASIC demands a continuous area for programs. This is why the screen is moved—otherwise, the video screen will be in the middle of your program. The same reason prevents the 3K RAM area from being used by BASIC when additional memory is added to the large expansion area. However, machine language programs can still use this area though.

The start of BASIC will begin at 4608 (\$1200), the video screen will start at 4096 (\$1000) and color memory will start at 37888 (\$9400).

See Section 3 for the formulas to use to calculate the screen start address. If you want your programs to work on any VIC memory configuration, your program must use these formulas in POKES and PEEKs to the screen. The best way to use this is at the beginning, set a variable to the start of screen memory and one to the start of color memory. Then, do any POKES or PEEKs to the screen relative to those variables. (Example: if C is the start of the screen, to put an 'A' on the first line on the 10th column of the screen, type: `POKEC+10,1`.)

GAME CONTROLLERS

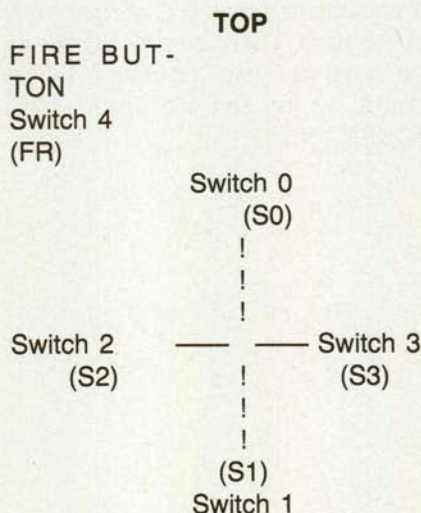
USING A JOYSTICK ON THE VIC

Like all other input and output, the joysticks are controlled using the VIC'S 6522 Versatile Interface Adapters (VIAs). The 6522 is a very versatile and complex device. Fortunately, it isn't necessary to delve deeply into the mysteries of the 6522 VIA to read the joysticks.

Each 6522 has two Input/Output ports, called port A and port B. Each of the ports has a control register attached, called the DATA DIRECTION REGISTER (DDR). This highly important register controls the direction of the port. By using this register you can use the port for input, output, or both at the same time. To set one bit of the port to output, set the corresponding bit of the Data Direction Register to 1. To set a bit of the port for input, set the corresponding bit of the DDR to 0. For example, to set bit 7 of port A to input, and the rest of the bits to output, poke 127 in the DDR for port A.

To read the joystick, one port (and one DDR) of each of the 6522 VIAs on the VIC must be used.

The joystick switches are arranged as follows:



Switch 0, Switch 1, Switch 2, and the Fire button can be read from VIA #1, which is located beginning at location \$9110. Switch 3 must be read from the other 6522 (VIA #2) which is located beginning at location \$9120.

Now, the key locations for the joystick are as follows:

HEX	DECIMAL	PURPOSE
9113	37139	Data direction register for I/O port A on VIA #1
9111	37137	Output register A Bit 2 Joy switch 0 Bit 3 Joy switch 1 Bit 4 Joy switch 2 Bit 5 Fire button
9122	37154	Data direction register for I/O port B on VIA #2
9120	37152	Output register B Bit 7 Joy switch 3

To read the joystick inputs, you first set the ports to input mode by setting the DDR to 0. This can be done by a POKE. Then the value of the switches can be read by two peeks. Sounds easy, right? There is only one problem . . . VIA#2 is also used for reading the keyboard. Setting the DDR can mess up the keyscan rather badly. So you have to make sure you restore the DDR to the original condition if you want to use the keyboard afterwards.

To make things really easy, you can use the following program. Lines 10 to 25 are initialization. The rest of the program, beginning at line 9000, can be called as a subroutine whenever you want to read the joystick.

```

10 DIM JS(2,2):POKE37139,0:DD=37154:PA=37137:
   PB=37152

20 FORI=0TO2:FORJ=0TO2:READJS (J,I):NEXTJ,I
25 DATA -23,-22,-21,-1,0,1,21,22,23

30 GOSUB9000:PRINT JS(X+1,Y+1):GOTO30

9000 POKEDD,127:S3=-((PEEK(PB)AND128)=0):
   POKEDD,255

9010 P=PEEK(PA):S1=-((PAND8)=0):S2=((PAND16)=0)
   :S0=((PAND4)=0)

9020 FR=-((PAND32)=0):X=S2+S3:Y=S0+S1:RETURN

```

The variables S0, S1, S2, and S3 will be 0 normally, and will be set to 1 (or -1) when the joystick is pointed in that direction. Two of

the variables will be set to 1 on diagonal moves. FR will be 1 when the firing button is pressed, 0 otherwise.

The AND function is used to pick out one bit of the joystick port. The bits are numbered from 7 (most significant bit) to 0 (least significant bit). By ANDing the 6522 port with a number whose value is a power of two, a single bit is selected. (For example, to pick bit 3, AND using 2,3 or 8).

The JS array in the program is set up to make moving around the screen using the joystick easy. The numbers in the DATA statement of line 25 can easily be changed for other purposes. For example . . .

To "decode" the joystick in this pattern:

TOP
FIRE

0
7 ! 1
6 — 8 — 2
5 ! 3
4

The data statement should be changed to:

25 DATA 7,0,1,6,8,2,5,4,3

USING PADDLES ON THE VIC

The paddles are read using both the VIC chip and the VIC's 6522 Versatile Interface Adapters (VIAs).

The values of the paddles are read through the VIC chip. There are two registers, one for each paddle, which will contain the current value of the paddle. This data will be in digitized form, as a value from 0 to 255.

The switches on each paddle are read from the VIA chips. Each VIA has two INPUT/OUTPUT ports, called PORT A and PORT B. Each of the ports has a control register attached, called the DATA DIRECTION REGISTER (DDR). This register controls the direction of the port. By using this register you can use the port for input, output, or both at the same time. To set one bit of the port to output, set the corresponding bit of the Data Direction Register to 1. To set a bit of the port for input, set the corresponding bit of the DDR to 0. For example, to set bit 7 of port A to input, and the rest of the bits to output, poke 127 in the DDR for port A.

To read the paddle switches, one port (and one DDR) of each of the 6522 VIAs on the VIC must be used.

The joystick switches are arranged as follows:

Paddle X ——— Paddle Y
(S2) (S3)

Switch 2 can be read from VIA #1, which is located beginning at location \$9110. Switch 3 must be read from the other 6522 (VIA #2) which is located beginning at location \$9120.

Now, the key locations for the paddle are as follows:

HEX	DECIMAL	PURPOSE
9008	36872	Digitized value of PADDLE X
9009	36873	Digitized value of PADDLE Y
9113	37139	Data direction register for I/O port A on VIA #1
9111	37137	Output register A Bit 4 PADDLE SWITCH X
9122	37154	Data direction register for I/O port B on VIA #2
9120	37152	Output register B Bit 7 PADDLE SWITCH Y

To read the paddle inputs, you first set the ports to input mode by setting the DDR to 0. This can be done by a POKE. Then the value of the switches can be read by two peeks. Sounds easy, right? There is only one problem . . . VIA#2 is also used for reading the keyboard. Setting the DDR can mess up the keyscan rather badly. So you have to make sure you restore the DDR to the original condition if you want to use the keyboard afterwards.

To make things really easy, you can use the following program. Lines 10 to 25 are initialization. The rest of the program, beginning at line 9000, can be called as a subroutine whenever you want to read the paddle.

```
10 POKE37139,0:DD=37154:PA=37137:PB=37152
20 PX=36872:PY=36873
30 GOSUB9000:PRINT PEEK(PX);PEEK(PY);X;Y:GOTO30
9000POKEDD,127:Y=-((PEEK(PB)AND128)=0):PO
KEDD,255
9010 X=-((PEEK(PA)AND16)=0):RETURN
```

The variables X and Y will be 0 normally, and will be set to 1 when that paddle button is pressed.

The AND function is used to pick out one bit of the paddle port. The bits are numbered from 7 (most significant bit) to 0 (least significant bit). By ANDing the 6522 port with a number whose value is a power of two, a single bit is selected. (For example, to pick bit 3, AND using 8.)

USING A LIGHT PEN ON THE VIC

One of the benefits of using the VIC chip as the controller for the VIC 20 is that it is easy to add certain input/output devices for games and educational software. It is as easy to add a light pen as it is to add game paddles and joysticks.

The principle behind the light pen is simple. Basically, the pen is a light detector, set to detect either the presence or absence of light. The television picture is not put on the screen all at once—rather, it is put on the screen one row at a time, scanning from left to right very quickly.

When the scan passes the area where the pen is, a signal is sent to the VIC chip. When this signal is received, the VIC chip, which keeps track of where the scan line is at any particular moment, will record the exact location of the scan in two registers, 36870 (9006 HEX) for the X direction and 36871 (9007) in the Y direction. You can read and use this information in your programs.

The light pen trigger is connected to pin 6 of the game port. The light pen trigger input can also be reached from pin 7 of the user port. Note that you can't use a joystick and a light pen at the same time, because the same line that is used as the light pen trigger input is used as the joystick fire button input (you would get false readings).

The VIC chip constantly keeps track of the scan position on the television in two registers. When the light pen trigger input is brought low, the VIC freezes the two registers. You can then read and use this information. After reading the two registers, the trigger line will be cleared, so that scan information can be placed again in the two registers.

RS-232 INTERFACE DESCRIPTION — BUILT-IN SOFTWARE

GENERAL OUTLINE

The VIC has a built-in RS-232 interface for connection to any RS-232 modem, printer, or other device. To connect the device to the VIC a cable is required, as well as some programming.

RS-232 on the VIC is standard RS-232 format, but the voltages are TTL levels (0 to 5V) rather than the normal RS-232 -12 to 12 volt range. The cable between the VIC and the RS-232 device should take care of the voltage conversions needed. The Commodore VIC RS-232 interface cartridge handles this properly.

The RS-232 interface software can be accessed from BASIC or from the KERNAL for machine language programming. RS-232 on the BASIC level uses the normal BASIC commands: OPEN, CLOSE, CMD, INPUT#, GET#, PRINT#, and the reserved variable ST. INPUT#(CHRIN for the machine language programmers in the audience) and GET#(GETIN) fetch data from the receiving buffer, while PRINT# (chrout) and CMD place data into the transmitting buffer. The use of these commands (and examples) will be described more fully later.

The RS-232 KERNAL byte/bit level handlers run under the control of the 6522 device timers and interrupts. The 6522 generates NMI requests for RS-232 processing. This allows background RS-232 processing to take place during BASIC and machine language programs. There are built-in hold-offs in the KERNAL cassette and serial bus routines to prevent disruption of data storage/transmission by the NMI's generated by the RS-232 routines. During cassette or serial bus activities data cannot be received from RS-232 devices. Because these hold-offs are only local (assuming care is taken in programming) no interference should result.

There are two buffers in the VIC RS-232 interface to help prevent loss of data when transmitting or receiving RS-232. The VIC 20 RS-232 KERNAL buffers consist of two first-in/first-out (FIFO) buffers, each 256 bytes long, at the top of memory. The OPENing of an RS-232 channel automatically allocates 512 bytes of memory for these buffers. If there is not enough free space beyond the end of your BASIC program no error message will be printed, and the end of your program will be destroyed. SO BE CAREFUL!

These buffers are automatically removed by the CLOSE command.

OPENING AN RS-232 CHANNEL

Only one RS-232 channel should be open at any time; a second OPEN statement will cause the buffer pointers to be reset. Any characters in either the transmit buffer or the received buffer will be lost.

Up to 4 characters may be sent in the filename field. The first two are the control and command register characters; the other two are reserved for future system options. Baud rate, parity, and other options can be selected through this feature.

No error-checking is done on the control word to detect a nonimplemented baud rate, so that any illegal control word will cause the system output to operate at a very slow rate (below 50 baud).

BASIC SYNTAX

OPENIf,2,0,"<control register><command register>"

If—Normal logical file id (1-255). If If>127 then linefeed follows carriage return.

<control register>—Single byte character (see Figure 1) (required to specify baud rate)

<command register>—Single byte character (see Figure 2) (this character is NOT required)

KERNAL ENTRY

OPEN (\$FFC0)—See KERNAL spec. for more information on entry conditions and instructions.

NOTE

IMPORTANT: In a BASIC program, the RS-232 OPEN command should be performed before using any variable or DIM statement, since an automatic CLR is performed when an RS-232 channel is OPENed (because of the allocation of 512 bytes at the top of memory.) Also remember that your program will be destroyed if 512 bytes of space are not available at the time of the OPEN statement.

GETTING DATA FROM RS-232 CHANNEL

When getting data, the VIC receiver buffer will hold 255 characters before a buffer overflow. This is indicated in the RS-232 status word (ST from BASIC, rsstat from machine language). If this occurs, all characters received during a full buffer condition are lost. Obviously it pays to keep the buffer as clear as possible.

If you wish to receive RS-232 data at high speeds (BASIC can only go so fast, especially considering garbage collects. This can cause the receiver buffer to overflow), you will have to use machine language routines to handle the data bursts.

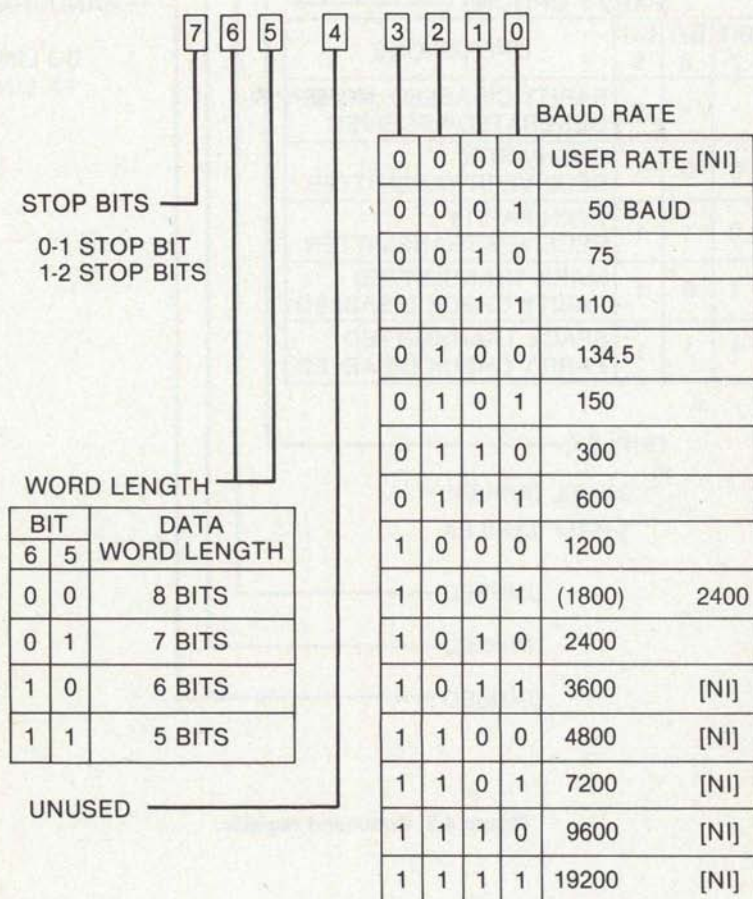


Figure 4-1. Control register.

BASIC SYNTAX

Recommended: GET#If,<string variable>

NOT Recommended: INPUT#If,<variable list>

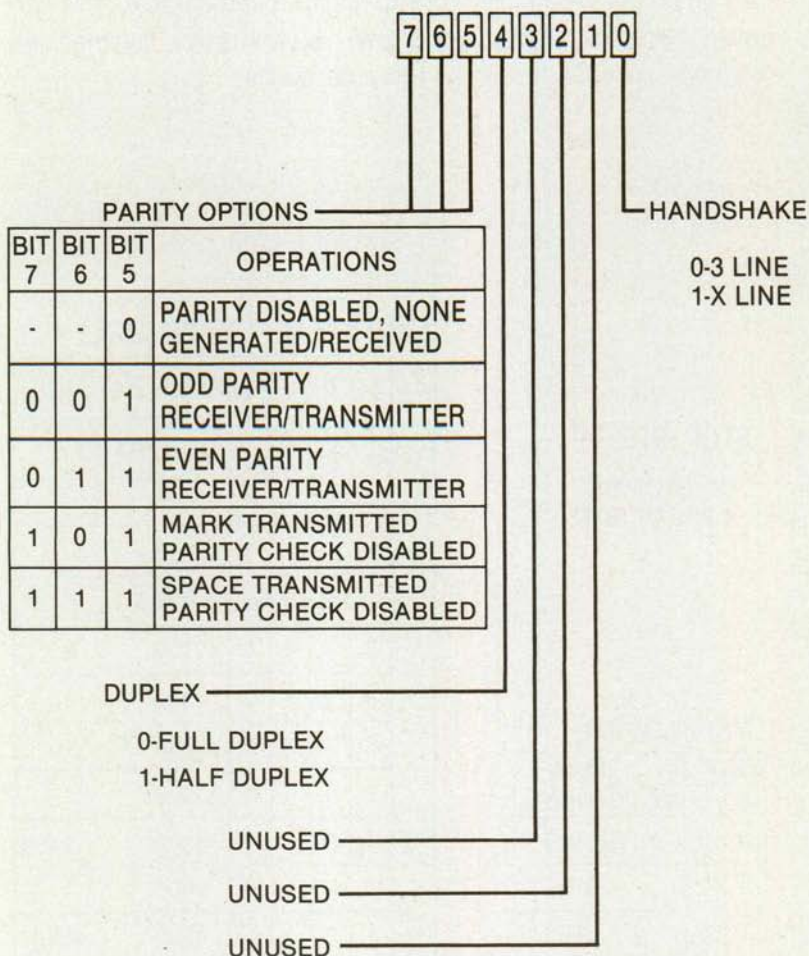


Figure 4-2. Command register.

KERNAL ENTRIES

CHKIN (\$FFC6)—See Section 3 for more information on entry and exit conditions.

GETIN (\$FFE4)—See Section 3 for more information on entry and exit conditions.

CHRIN (\$FFCF)—See Section 3 for more information on entry and exit conditions.

NOTES

If the word length is less than 8 bits, all unused bit(s) will be assigned a value of zero.

If a GET# does not find any data in the buffer, the character "" (a null) is returned.

If INPUT# is used, then the system will hang until a non-null character and a following carriage return is received. Thus, if the CTS or DSR line(s) disappear during character INPUT#, the system will hang in a RESTORE-only state. This is why the INPUT# and CHRIN routines are NOT recommended.

The routine CHKIN handles the x-line handshake which follows the EIA standard (August 1979) for RS-232-C interfaces. (The RTS, and DCD lines are implemented with the VIC computer defined as the Data Terminal device.)

SENDING DATA TO AN RS-232 CHANNEL

When sending data, the output buffer can hold 256 characters before a full buffer hold-off occurs. The system will wait in the CHROUT routine until transmission is allowed or the RUN/STOP-RESTORE keys are used to recover the system through a WARM START.

BASIC SYNTAX

CMD If—acts same as in BASIC spec.

PRINT#If,<variable list>

KERNAL ENTRIES

CHKOUT (\$FFC9)—See Section 3 for more information on entry and exit conditions.

CHROUT (\$FFD2)—See Section 3 for more information on entry conditions.

NOTES

IMPORTANT: There is no carriage-return delay built into the output channel so a normal RS-232 printer cannot correctly print, unless some form of hold-off (asking the VIC to wait) or internal buffering is implemented by the printer. The hold-off can easily be implemented in your program. If a CTS (x-line) handshake is implemented, the VIC buffer will fill and hold off more output until transmission is allowed by the RS-232 device.

The routine CHKOUT handles the x-line handshake, which follows the EIA standard (August 1979) for RS-232-C interfaces. The RTS, and DCD lines are implemented with the VIC defined as the Data Terminal Device.

CLOSING AN RS-232 DATA CHANNEL

Closing an RS-232 file discards all data in the buffers at the time of execution (whether or not it had been transmitted or printed out), stops all RS-232 transmitting and receiving, sets the RTS and Sout lines high, and removes both RS-232 buffers.

BASIC SYNTAX

CLOSE If

KERNAL ENTRY

CLOSE (\$FFC3)—See Section 3 for more information on entry and exit conditions.

NOTE

Care should be taken to ensure all data is transmitted before closing the channel. A way to check this from BASIC is:

```
100 IF ST=0 AND (PEEK(37151) AND 64)=1 GOTO 100
110 CLOSE if
```

Table 4-1. USER-PORT LINES

PIN 6522		DESCRIPTION MODES	(6522 DEVICE #1 loc \$9110-911F)		
ID	ID		EIA	ABV	IN/ OUT
C	PB0	RECEIVED DATA	(BB) Sin	IN	1 2
D	PB1	REQUEST TO SEND	(CA) RTS	OUT	1*2
E	PB2	DATA TERMINAL READY	(CD) DTR	OUT	1*2
F	PB3	RING INDICATOR	(CE) RI	IN	3
H	PB4	RECEIVED LINE SIGNAL	(CF) DCD	IN	2
J	PB5	UNASSIGNED	() XXX	IN	3
K	PB6	CLEAR TO SEND	(CB) CTS	IN	3
L	PB7	DATA SET READY	(CC) DSR	IN	2
B	CB1	RECEIVED DATA	(BB) Sin	IN	1 2
M	CB2	TRANSMITTED DATA	(BA) Sout	OUT	1 2
A	GND	PROTECTIVE GROUND	(AA) GND		1 2
N	GND	SIGNAL GROUND	(AB) GND		1 2 3

MODES

- 1)—3-LINE INTERFACE (Sin,Sout,GND)
 - 2)—X-LINE INTERFACE (Full handshaking)
 - 3)—USER AVAILABLE ONLY (Unused/unimplemented in code.)
- *—These lines are held high during 3-LINE mode.

*Note: PB6 CLEAR TO SEND is not implemented and must be read with a short machine language routine.

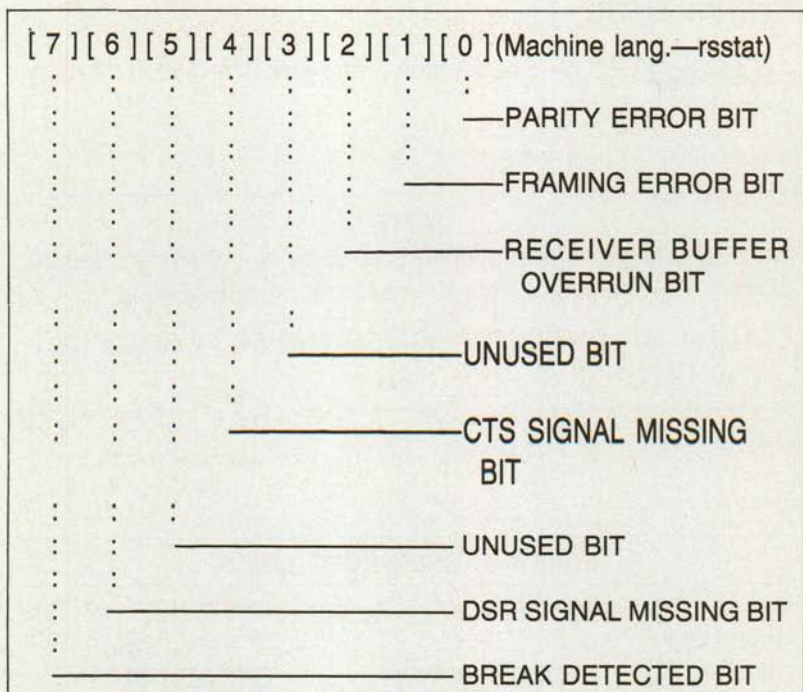


Figure 4-3. RS-232 Status Register

NOTES

If the BIT = 0, then no error has been detected.

The RS-232 status register can be read from BASIC using the variable ST.

If ST is read by BASIC or by using the KERNAL READST routine the RS-232 status word is cleared upon exit. If multiple uses of the STATUS word are necessary the ST should be assigned to another variable, i.e.

SR = ST: REM ASSIGNS ST TO SR

The RS-232 status is read (and cleared) only when the RS-232 channel was the last external I/O used.

SAMPLE BASIC PROGRAM

```
10 REM THIS PROGRAM SENDS AND RECEIVES DATA
TO/FROM A SILENT 700 TERMINAL MODIFIED FOR PET
ASCII
```

```
20 REM TI SILENT 700 SET-UP: 300 BAUD, 7-BIT ASCII,
MARK PARITY, FULL DUPLEX
```

```

30 REM SAME SET-UP AT COMPUTER USING 3-LINE
INTERFACE
100 OPEN 2,2,3,CHR$(6+32)+CHR$(32+128): REM OPEN
THE CHANNEL
110 GET#2,A$: REM TURN ON THE RECEIVER CHANNEL
(TOSS A NULL)
200 REM MAIN LOOP
210 GET B$: REM GET FROM COMPUTER KEYBOARD
220 IF B$<>"" THEN PRINT#2,B$,: REM IF A KEY
PRESSED, SEND TO TERMINAL
230 GET#2,C$: REM GET A KEY FROM THE TERMINAL
240 PRINT B$;C$,: REM PRINT ALL INPUTS TO THE
COMPUTER SCREEN
250 SR=ST: IF SR=0 THEN 200: REM CHECK STATUS, IF
GOOD THEN CONTINUE
300 REM ERROR REPORTING
310 PRINT "ERROR: ";
320 IF SR AND 1 THEN PRINT "PARITY"
330 IF SR AND 2 THEN PRINT "FRAME"
340 IF SR AND 4 THEN PRINT "RECEIVER BUFFER FULL"
350 IF SR AND 128 THEN PRINT "BREAK"
360 IF (PEEK(37151) AND 64)=1 THEN 360: REM WAIT
UNTIL ALL CHARS TRANSMITTED
370 CLOSE 2: END

```

RECEIVER/TRANSMITTER BUFFER BASE LOCATION POINTERS

\$00F7-RIBUF A two byte pointer to the Receiver Buffer base location.

\$00F9-ROBUF A two byte pointer to the Transmitter Buffer base location.

The two locations above are set up by the KERNAL OPEN routine, each pointing to a different 256 byte buffer. They are de-allocated by writing a zero into the high order bytes, (\$00F8 and \$00F9), which is done by the KERNAL CLOSE entry. They may also be allocated/de-allocated by the machine language programmer for his/her own purposes, removing/creating only the buffer(s) required. Both the OPEN and CLOSE routines will not notice that their jobs might have been done already. When using a machine language program that allocates these buffers, care must be taken to make sure that the top of memory pointers stay correct, especially if BASIC programs are expected to run at the same time.

ZERO-PAGE MEMORY LOCATIONS AND USAGE FOR RS-232 SYSTEM INTERFACE

\$00A7-INBIT Receiver input bit temp storage.

\$00A8-BITCI Receiver bit count in.

\$00A9-RINONE Receiver flag Start bit check.

\$00AA-RIDATA Receiver byte buffer/assembly location.

\$00AB-RIPRTY Receiver parity bit storage.

\$00B4-BITTS Transmitter bit count out.

\$00B5—NXTBIT Transmitter next bit to be sent

\$00B6-RODATA Transmitter byte buffer/disassembly location.

All the above zero page locations are used locally and are only given as a guide to understand the associated routines. These cannot be used directly by the BASIC or KERNAL level programmer to do RS-232 type things. The system RS-232 routines must be used.

NONZERO-PAGE MEMORY LOCATIONS AND USAGE FOR RS-232 SYSTEM INTERFACE

General RS-232 storage:

\$0293-M51CTR Pseudo 6551 control register (see Figure 4-1).

\$0294-M51CDR Pseudo 6551 command register (see Figure 4-2).

\$0295-M51AJB Two bytes following the control and command registers in the file name field. (For future use.)

\$0297-RSSTAT The RS-232 status register (see Figure 4-3).

\$0298-BITNUM The number of bits to be sent/received.

\$0299-BAUDOF Two bytes that are equal to the time of one bit cell. (Based on system clock/ baud rate.)

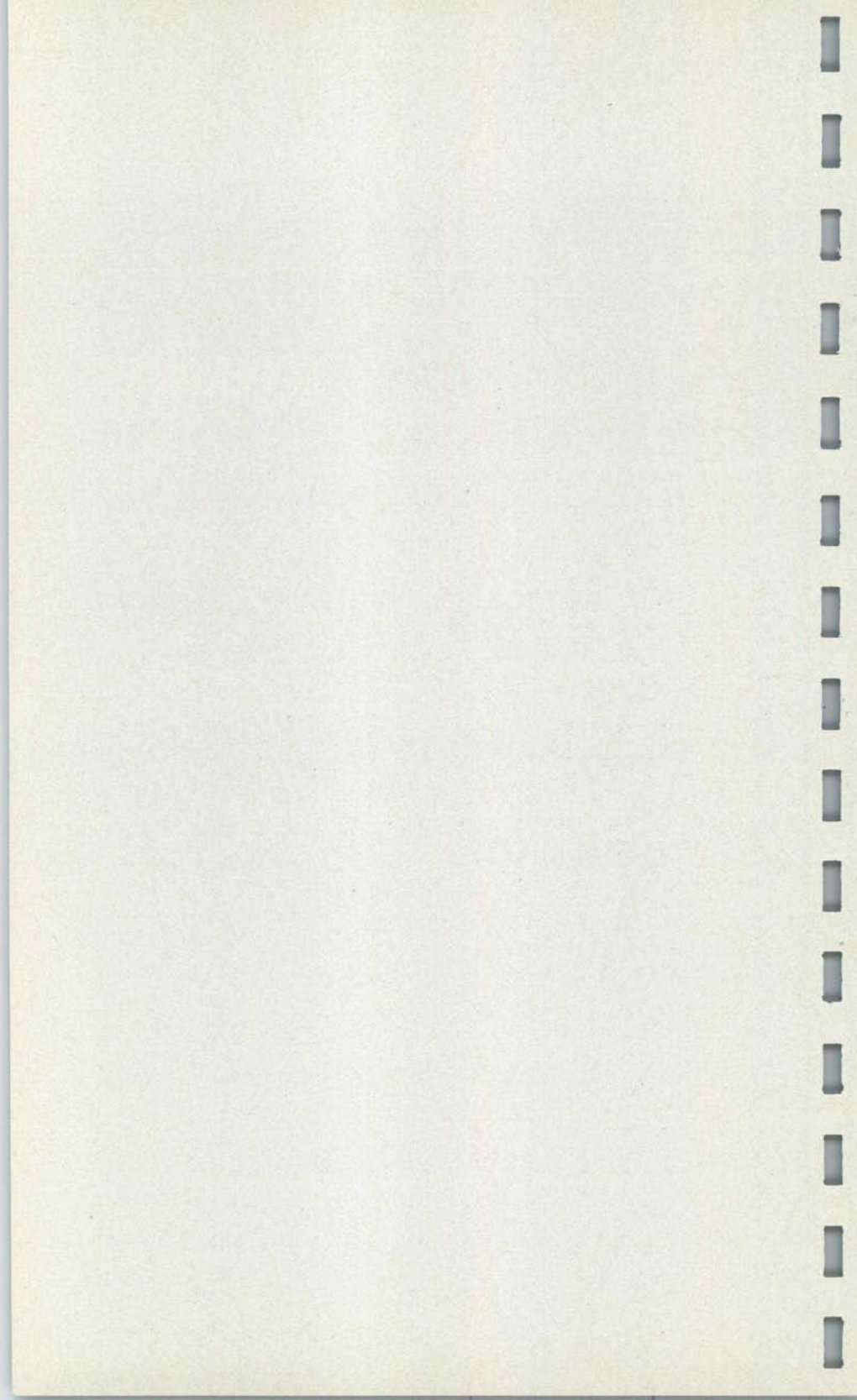
\$029B-RIDBE The byte index to the end of the receiver FIFO buffer.

\$029C-RIDBS The byte index to the start of the receiver FIFO buffer.

\$029D-RODBS The byte index to the start of the transmitter FIFO buffer.

\$029E-RODBE The byte index to the end of the transmitter FIFO buffer.

APPENDICES


















APPENDIX A

ABBREVIATIONS FOR BASIC KEYWORDS

As a time saver when typing in programs and commands, VIC BASIC allows the user to abbreviate most keywords. The abbreviation for the word PRINT is a question mark. The abbreviations for the other words are made by typing the first one or two letters of the keyword, followed by the SHIFTed next letter of the word. If the abbreviations are used in a program line, the keyword will LIST in the longer form. Note that some of the keywords when abbreviated include the first parenthesis, and others do not.

Command	Abbreviation	Looks like this on screen	Command	Abbreviation	Looks like this on screen
ABS	A SHIFT B	A	INPUT#	I SHIFT N	I
AND	A SHIFT	A	LET	L SHIFT E	L
ASC	A SHIFT S	A	LEFT\$	LE SHIFT F	LE
ATN	A SHIFT T	A	LIST	L SHIFT I	L
CHRS	C SHIFT H	C	LOAD	L SHIFT O	L
CLOSE	CL SHIFT O	CL	MIDS	M SHIFT I	M
CLR	C SHIFT L	C	NEXT	N SHIFT <	N
CMD	C SHIFT M	C	NOT	N SHIFT O	N
CONT	C SHIFT O	C	OPEN	O SHIFT P	O
DATA	D SHIFT A	D	PEEK	P SHIFT E	P
DEF	D SHIFT E	D	POKE	P SHIFT O	P
DIM	D SHIFT I	D	PRINT	?	?
END	E SHIFT N	E	PRINT#	P SHIFT R	P
EXP	E SHIFT X	E	READ	R SHIFT E	R
FOR	F SHIFT O	F	RESTORE	RE SHIFT S	RE
FRE	F SHIFT R	F	RETURN	RE SHIFT T	RE
GET	G SHIFT E	G	RIGHT\$	R SHIFT I	R
GOSUB	GO SHIFT S	GO	RND	R SHIFT N	R
GOTO	SHIFT O	G	RUN	R SHIFT	R

Command	Abbreviation	Looks like this on screen
SAVE	S SHIFT A	S 
SGN	S SHIFT G	S 
SIN	S SHIFT I	S 
SPC(S SHIFT P	S 
SQR	S SHIFT Q	S 
STEP	ST SHIFT E	ST 
STOP	S SHIFT T	S 
STR\$	ST SHIFT R	ST 

Command	Abbreviation	Looks like this on screen
SYS	S SHIFT Y	S 
TAB	T SHIFT A	T 
THEN	T SHIFT H	T 
USR	U SHIFT S	U 
VAL	V SHIFT A	V 
VERIFY	V SHIFT E	V 
WAIT	W SHIFT A	W 

COLOR CODE TABLE

Following are the various colors the VIC can display . . . note that colors 8-15 can only be used as a SCREEN COLOR or an AUXILIARY COLOR (see pg. 93 for explanation of auxiliary colors used in MULTICOLOR MODE). As an example, these color numbers are used to POKE a color into "color memory" when coloring characters POKEd to the screen. If you POKE 7680, 81 this places a "ball" on the screen but it will be "invisible" until you add the color by typing POKE 38400, 0 (BLACK). Try POKE38400,2 for RED, etc. (Numbers 8-15 cannot be used as character colors)

BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7
ORANGE	8
LT. ORANGE	9
PINK	10
LT. CYAN	11
LT. PURPLE	12
LT. GREEN	13
LT. BLUE	14
LT. YELLOW	15

APPENDIX B

SCREEN & BORDER COLOR COMBINATIONS

You can change the screen and border colors of the VIC anytime, in or out of a program, by typing

POKE 36879, X

where X is one of the numbers shown in the chart below. POKE 36879, 27 returns the screen to the normal color combination, which is a CYAN border and white screen.

Try typing POKE 36879,8. Then type **CTRL** **WHT** and you have white letters on a totally black screen! Try some other combinations. This POKE command is a quick and easy way to change screen colors in a program.

SCREEN	BORDER							
	BLK	WHT	RED	CYAN	PUR	GRN	BLU	YEL
BLACK	8	9	10	11	12	13	14	15
WHITE	24	25	26	27	28	29	30	31
RED	40	41	42	43	44	45	46	47
CYAN	56	57	58	59	60	61	62	63
PURPLE	72	73	74	75	76	77	78	79
GREEN	88	89	90	91	92	93	94	95
BLUE	104	105	106	107	108	109	110	111
YELLOW	120	121	122	123	124	125	126	127
ORANGE	136	137	138	139	140	141	142	143
LT. ORANGE	152	153	154	155	156	157	158	159
PINK	168	169	170	171	172	173	174	175
LT. CYAN	184	185	186	187	188	189	190	191
LT. PURPLE	200	201	202	203	204	205	206	207
LT. GREEN	216	217	218	219	220	221	222	223
LT. BLUE	232	233	234	235	236	237	238	239
LT. YELLOW	248	249	250	251	252	253	254	255

APPENDIX C

TABLE OF MUSICAL NOTES

APPROX. NOTE	VALUE	APPROX. NOTE	VALUE
C	135	G	215
C#	143	G#	217
D	147	A	219
D#	151	A#	221
E	159	B	223
F	163	C	225
F#	167	C#	227
G	175	D	228
G#	179	D#	229
A	183	E	231
A#	187	F	232
B	191	F#	233
C	195	G	235
C#	199	G#	236
D	201	A	237
D#	203	A#	238
E	207	B	239
F	209	C	240
F#	212	C#	241

SPEAKER COMMANDS:	WHERE X CAN BE:	FUNCTION:
POKE 36878, X	0 to 15	sets volume
POKE 36874, X	128 to 255	plays tone
POKE 36875, X	128 to 255	plays tone
POKE 36876, X	128 to 255	plays tone
POKE 36877, X	128 to 255	plays "noise"

APPENDIX D

SCREEN DISPLAY CODES

The following chart lists all of the characters built-in to the VIC 20 character sets. It shows which numbers should be POKEd into screen memory (locations 7680 to 8185) to get a desired character. Also, it shows what character corresponds to a number PEEKed from the screen.

The two character sets are available, but only one set at a time. This means that you cannot have characters from one set on the screen at the same time you have characters from the other set displayed. The sets are switched by holding down the SHIFT and COMMODORE keys simultaneously. This actually changes the 2 bit in memory location 36869, which means that the statement POKE 36869, 240 will set the character set to upper case, and POKE 36869, 242 switches to lower case.

If you want to do some serious animation, you will find that it is easier to control objects on the screen by POKEing them into screen memory (and erasing them by POKEing a 32, which is the code for a blank space, into the same memory location), than by PRINTing to the screen by using cursor control characters.

Any number shown on the chart may also be displayed in REVERSE. Reverse characters are not shown, but the reverse of any character may be obtained by adding 128 to the numbers shown.

NOTE: SEE SCREEN MEMORY MAP APPENDIX E

If you want to display a heart at screen location 7800, find the number of the character you want to display there (in this case a heart) in this chart . . . the number for the heart is 83 . . . then type in a POKE statement with the number of the screen location (7800) and the number of the symbol (83) like this:

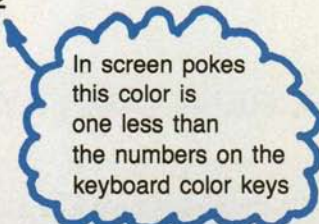
POKE 7800,83

A white heart should appear in the middle area of the screen. Note that it will be invisible if the screen is white. Try changing the position by changing the larger number, or type in different symbols using the numbers from the chart.

If you want to change the COLOR of the symbol being displayed, consult the Color Codes Memory Map in Appendix E which lists the COLOR NUMBERS for EACH MEMORY LOCATION. In other words, to get a different colored symbol at a particular location, this requires another POKE command.

For example, to get a red heart, type the following:

POKE 38520, 2






































































In screen pokes
this color is
one less than
the numbers on the
keyboard color keys

This changes the color of the symbol at location 7800 to red. If you had a different symbol there, that symbol would now be red. You can display any character in any of the available colors by combining these two charts. These POKE commands can be added in your programs and are very effective especially in animation—and also provide a means to PEEK at certain locations if you are doing sophisticated programming such as bouncing a ball, which requires this information.

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	R	r	18	\$		36
A	a	1	S	s	19	%		37
B	b	2	T	t	20	&		38
C	c	3	U	u	21	'		39
D	d	4	V	v	22	(40
E	e	5	W	w	23)		41
F	f	6	X	x	24	*		42
G	g	7	Y	y	25	+		43
H	h	8	Z	z	26	,		44
I	i	9	[27	-		45
J	j	10	£		28	.		46
K	k	11]		29	/		47
L	l	12	↑		30	0		48
M	m	13	←		31	1		49
N	n	14	SPACE		32	2		50
O	o	15	!		33	3		51
P	p	16	"		34	4		52
Q	q	17	#		35	5		53

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
6		54		O	79			104
7		55		P	80			105
8		56		Q	81			106
9		57		R	82			107
:		58		S	83			108
;		59		T	84			109
<		60		U	85			110
=		61		V	86			111
>		62		W	87			112
?		63		X	88			113
		64		Y	89			114
	A	65		Z	90			115
	B	66			91			116
	C	67			92			117
	D	68			93			118
	E	69			94			119
	F	70			95			120
	G	71	SPACE		96			121
	H	72			97			122
	I	73			98			123
	J	74			99			124
	K	75			100			125
	L	76			101			126
	M	77			102			127
	N	78			103			

Codes from 128-255 are reversed images of codes 0-127.

APPENDIX E

SCREEN MEMORY MAPS

Use this appendix to find the memory location of any position on the screen. Just find the position in the grid and add the numbers on the row and column together. For example, if you want to poke the graphic "ball" character onto the center of the screen, add the numbers at the edge of row 11 and column 11 ($7900 + 10$) for a total of 7910. If you poke the code for a ball (81, see Appendix D) into location 7910 by typing `POKE 7910,81`, a white ball appears on the screen. To change the color of the ball (or other character), find the corresponding position on the color codes memory map, add the row and column numbers together ($38620 + 10$, or 38630) for the color code and type a second poke statement. For example, if you poke a color code into this location, `POKE 38630,3` the ball will change color to cyan. Note that when POKEing, the character color numbers are one less than the numbers on the color keys—as shown below.

Abbreviated List of Color Codes:

Code	Color
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7680																						
7702																						
7724																						
7746																						
7768																						
7790																						
7812																						
7834																						
7856																						
7878																						
7900																						
7922																						
7944																						
7966																						
7988																						
8010																						
8032																						
8054																						
8076																						
8098																						
8120																						
8142																						
8164																						

Screen Character Codes












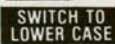




	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
38400																						
38422																						
38444																						
38466																						
38488																						
38510																						
38532																						
38554																						
38576																						
38598																						
38620																						
38642																						
38664																						
38686																						
38708																						
38730																						
38752																						
38774																						
38796																						
38818																						
38840																						
38862																						
38884																						

Color Codes Memory Map

APPENDIX F

ASCII AND CHR\$ CODES

This appendix shows you what characters will appear if you PRINT CHR\$(X), for all possible values of X. It will also show the values obtained by typing PRINT ASC ("x") where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper/lower case, and printing character-based commands (like switch to upper/lower case) that could not be enclosed in quotes.

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	0		22	,	44	B	66
	1		23	-	45	C	67
	2		24	.	46	D	68
	3		25	/	47	E	69
	4		26	0	48	F	70
	5		27	1	49	G	71
	6		28	2	50	H	72
	7		29	3	51	I	73
DISABLES  	8		30	4	52	J	74
ENABLES  	9		31	5	53	K	75
	10		32	6	54	L	76
	11	!	33	7	55	M	77
	12	"	34	8	56	N	78
	13	#	35	9	57	O	79
	14	\$	36	:	58	P	80
	15	%	37	;	59	Q	81
	16	&	38	<	60	R	82
	17	.	39	=	61	S	83
	18	(40	>	62	T	84
	19)	41	?	63	U	85
	20	*	42	@	64	V	86
	21	+	43	A	65	W	87

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
X	88		114	f8	140		166
Y	89		115		141		167
Z	90		116		142		168
[91		117		143		169
£	92		118		144		170
]	93		119		145		171
↑	94		120		146		172
←	95		121		147		173
	96		122		148		174
	97		123		149		175
	98		124		150		176
	99		125		151		177
	100		126		152		178
	101		127		153		179
	102		128		154		180
	103		129		155		181
	104		130		156		182
	105		131		157		183
	106		132		158		184
	107	f1	133		159		185
	108	f3	134		160		186
	109	f5	135		161		187
	110	f7	136		162		188
	111	f2	137		163		189
	112	f4	138		164		190
	113	f6	139		165		191

CODES
CODES
CODE

192-223
224-254
255

SAME AS
SAME AS
SAME AS

96-127
160-190
126

ASCII Character Codes (decimal)*

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NULL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	bslash
007	BEL	050	2	093]
008	BS	051	3	094	up arrow
009	HT	052	4	095	back arr
010	LF	053	5	096	space
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	\$	079	O	122	z
037	%	080	P	123	;
038	&	081	Q	124	<
039	'	082	R	125	=
040	(083	S	126	>
041)	084	T	127	DEL
042	*	085	U		

*VIC character codes differ from ASCII codes. This table is provided as a reference for ASCII/VIC conversions.

APPENDIX G

DERIVING MATHEMATICAL FUNCTIONS

Functions that are not intrinsic to VIC BASIC may be calculated as follows:

FUNCTION	VIC BASIC EQUIVALENT
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X^2 + 1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X^2 + 1)) + \pi/2$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1))$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X^2 - 1)) + (\text{SGN}(X) - 1) * \pi/2$
INVERSE COTANGENT	$\text{ARCOT}(X) = \text{ATN}(X) + \pi/2$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = \text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X))^2 + 1$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X))^2 + 1$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X^2 + 1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X^2 - 1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1 + X)/(1 - X))/2$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X^2 + 1) + 1/X))$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X) * \text{SQR}(X^2 + 1/x))$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X + 1)/(X - 1))/2$

APPENDIX H

ERROR MESSAGES

This appendix contains a complete list of the error messages generated by the VIC, with a description of the causes.

BAD DATA String data was received from an open file, but the program was expecting numeric data.

BAD SUBSCRIPT The program was trying to reference an element of an array whose number is outside of the range specified in the DIM statement.

CAN'T CONTINUE The CONT command will not work, either because the program was never RUN, there has been an error, or a line has been edited.

DEVICE NOT PRESENT The required I/O device was not available for an OPEN, CLOSE, CMD, PRINT#, INPUT#, or GET#.

DIVISION BY ZERO Division by zero is a mathematical oddity and not allowed.

EXTRA IGNORED Too many items of data were typed in response to an INPUT statement. Only the first few items were accepted.

FILE NOT FOUND If you were looking for a file on tape, and END-OF-TAPE marker was found. If you were looking on disk, no file with that name exists.

FILE NOT OPEN The file specified in a CLOSE, CMD, PRINT#, INPUT#, or GET#, must first be OPENed.

FILE OPEN An attempt was made to open a file using the number of an already open file.

FORMULA TOO COMPLEX The string expression being evaluated should be split into at least two parts for the system to work with.

ILLEGAL DIRECT The INPUT statement can only be used within a program, and not in direct mode.

ILLEGAL QUANTITY A number used as the argument of a function or statement is out of the allowable range.

LOAD There is a problem with the program on tape.

NEXT WITHOUT FOR This is caused by either incorrectly nesting loops or having a variable name in a NEXT statement that doesn't correspond with one in a FOR statement.

NOT INPUT FILE An attempt was made to INPUT or GET data from a file which was specified to be for output only.

NOT OUTPUT FILE An attempt was made to PRINT data to a file which was specified as input only.

OUT OF DATA A READ statement was executed but there is no data left unREAD in a DATA statement.

OUT OF MEMORY There is no more RAM available for program or variables. This may also occur when too many FOR loops have been nested, or when there are too many GOSUBs in effect.

OVERFLOW The result of a computation is larger than the largest number allowed, which is $1.70141884E+38$.

REDIM'D ARRAY An array may only be DIMensioned once. If an array variable is used before that array is DIM'd, an automatic DIM operation is performed on that array setting the number of elements to ten, and any subsequent DIMs will cause this error.

REDO FROM START Character data was typed in during an INPUT statement when numeric data was expected. Just re-type the entry so that it is correct, and the program will continue by itself.

RETURN WITHOUT GOSUB A RETURN statement was encountered, and no GOSUB command has been issued.

STRING TOO LONG A string can contain up to 255 characters.

SYNTAX A statement is unrecognizable by the VIC. A missing or extra parenthesis, misspelled keywords, etc.

TYPE MISMATCH This error occurs when a number is used in place of a string, or vice-versa.

UNDEF'D FUNCTION A user defined function was referenced, but it has never been defined using the DEF FN statement.

UNDEF'D STATEMENT An attempt was made to GOTO or GOSUB or RUN a line number that doesn't exist.

VERIFY The program on tape or disk does not match the program currently in memory.

APPENDIX I

CONVERTING PROGRAMS TO VIC 20 BASIC

If you have programs written in a BASIC other than VIC 20 BASIC, some minor adjustments may be necessary before running them with VIC 20 BASIC. The following paragraphs specify things to look for when converting BASIC programs.

String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as `DIM A$(I,J)`, which dimensions a string array for J elements of length I, should be converted to the VIC 20 BASIC statement `DIM A$(J)`.

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for VIC 20 BASIC string concatenation.

In VIC 20 BASIC, the `MID$`, `RIGHT$`, and `LEFT$` functions are used to take substrings of strings. Forms such as `A$(I)` to access the Ith character in A\$, or `A$(I,J)` to take a substring of A\$ from position I to position J, must be changed as follows:

Other BASIC	VIC 20 BASIC
-------------	--------------

<code>A\$(I) = X\$</code>	<code>A\$ = LEFT\$(A\$,I-1) + X\$ + MID\$(A\$,I+1)</code>
---------------------------	---

<code>A\$(I,J) = X\$</code>	<code>A\$ = LEFT\$(A\$,I-1) + X\$ + MID\$(A\$,J+1)</code>
-----------------------------	---

Multiple Assignments

To set B and C equal to zero, some BASICs allow statements of the form:

```
10 LET B=C=0
```

VIC 20 BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (\) to separate multiple statements on a line. With VIC-20 BASIC, be sure all statements on a line are separated by a colon (:).

Mat Functions

Programs using the MAT functions available in some BASICs must be rewritten using `FOR . . . NEXT` loops to execute properly.

Character Tokens

To conserve user space, BASIC keywords are translated to 1-character tokens. The token values are shown in the following table.

TOKENS FOR VIC 20 BASIC

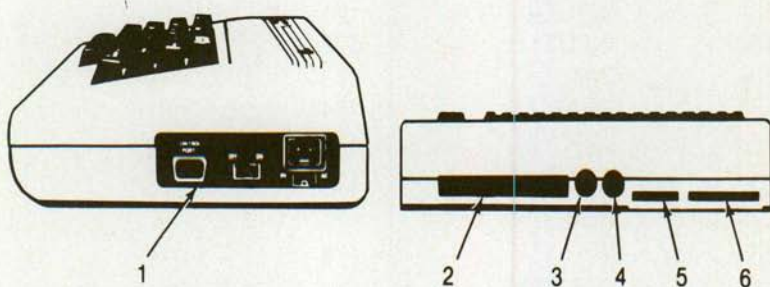
Token	BASIC keyword	Token	BASIC keyword
128	END	167	THEN
129	FOR	168	NOT
130	NEXT	169	STEP
131	DATA	170	+
132	INPUT#	171	-
133	INPUT	172	*
134	DIM	173	/
135	READ	174	
136	LET	175	AND
137	GOTO	176	OR
138	RUN	177	>
139	IF	178	=
140	RESTORE	179	<
141	GOSUB	180	SGN
142	RETURN	181	INT
143	REM	182	ABS
144	STOP	183	USR
145	ON	184	FRE
146	WAIT	185	POS
147	LOAD	186	SQR
148	SAVE	187	RND
149	VERIFY	188	LOG
150	DEF	189	EXP
151	POKE	190	COS
152	PRINT#	191	SIN
153	PRINT	192	TAN
154	CONT	193	ATN
155	LIST	194	PEEK
156	CLR	195	LEN
157	CMD	196	STR\$
158	SYS	197	VAL
159	OPEN	198	ASC
160	CLOSE	199	CHR\$
161	GET	200	LEFT\$
162	NEW	201	RIGHT\$
163	TAB (202	MID\$
164	TO	203	GO
165	FN	204	
166	SPC ((1) ?SYNTAX ERROR

Note: (1) The token after used token produces this error when listed.

APPENDIX J

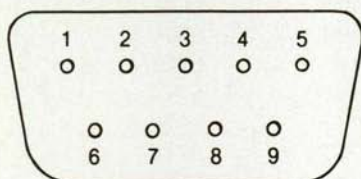
PINOUTS FOR INPUT/OUTPUT DEVICES

Here is a picture of the I/O ports on the VIC:



- | | |
|---------------------|----------------------|
| 1) Game I/O | 4) Serial I/O (disk) |
| 2) Memory Expansion | 5) Cassette |
| 3) Audio and Video | 6) User Port (modem) |

Game I/O



PIN #	TYPE	NOTE
1	JOYØ	MAX.100mA
2	JOY1	
3	JOY2	
4	JOY3	
5	POT Y	
6	LIGHT PEN	
7	+5V	
8	GND	
9	POT X	

Memory Expansion



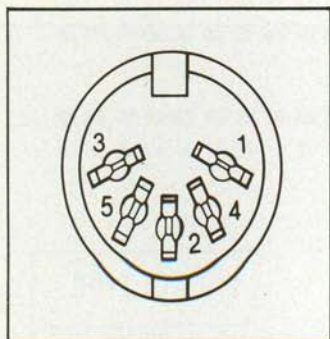
PIN #	TYPE
1	GND
2	CD0
3	CD1
4	CD2
5	CD3
6	CD4
7	CD5
8	CD6
9	CD7
10	<u>BLK1</u>
11	<u>BLK2</u>

PIN #	TYPE
12	<u>BLK3</u>
13	<u>BLK5</u>
14	<u>RAM1</u>
15	<u>RAM2</u>
16	<u>RAM3</u>
17	VR/W
18	CR/W
19	IRQ
20	NC
21	+5V
22	GND

PIN #	TYPE
A	GND
B	CA0
C	CA1
D	CA2
E	CA3
F	CA4
H	CA5
J	CA6
K	CA7
L	CA8
M	CA9

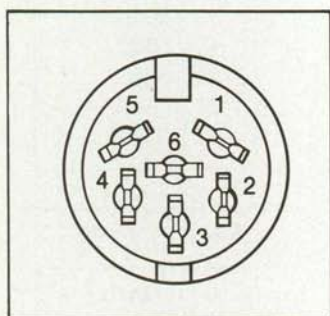
PIN #	TYPE
N	CA10
P	CA11
R	CA12
S	CA13
T	I/O2
U	I/O3
V	<u>S02</u>
W	<u>NMI</u>
X	<u>RESET</u>
Y	NC
Z	GND

Audio/Video



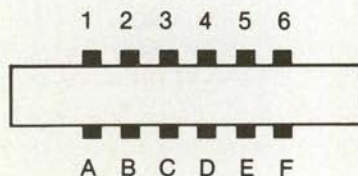
PIN #	TYPE	NOTE
1	+5V REG	10mA MAX
2	GND	
3	AUDIO	
4	VIDEO LOW	
5	VIDEO HIGH	

Serial I/O

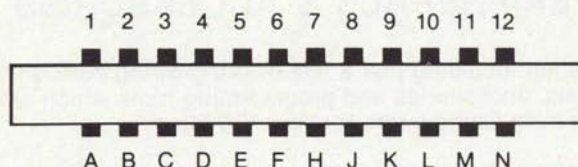


PIN #	TYPE
1	SERIAL SRQ IN
2	GND
3	SERIAL ATN IN/OUT
4	SERIAL CLK IN/OUT
5	SERIAL DATA IN/OUT
6	RESET

Cassette



PIN #	TYPE
A-1	GND
B-2	+5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SWITCH



PIN #	TYPE	NOTE	PIN #	TYPE	NOTE
1	GND	100mA MAX.	A	GND	
2	+5V		B	CB1	
3	RESET		C	PB0	
4	JOY0		D	PB1	
5	JOY1		E	PB2	
6	JOY2		F	PB3	
7	LIGHT PEN		H	PB4	
8	CASSETTE SWITCH		J	PB5	
9	SERIAL ATN IN	100mA MAX.	K	PB6	
10	+9V		L	PB7	
11	+9V		M	CB2	
12	GND		N	GND	

APPENDIX K

VIC PERIPHERALS & ACCESSORIES

Here is a list including just a few of the growing number of peripherals, accessories and programming tools which are available from Commodore for your VIC 20:

COMMODORE DATASSETTE ... for loading your own programs and replaying inexpensive pre-recorded tape programs.

VIC 1540 SINGLE DISK DRIVE ... stores up to 170K of data on a floppy diskette, for fast, high-capacity data storage and retrieval.

VIC GRAPHIC PRINTER ... 80 column dot matrix printer for making paper printouts; prints VIC graphics, letters, numbers and programmable characters.

VICMODEM ... Commodore's exclusive "affordable" modem on cartridge turns the VIC into a telecommunications terminal. Originate/answer, direct connect, 300 baud. VICTERM I tape included.

VIC 3K MEMORY EXPANDER ... 3K memory expansion on cartridge.

VIC 8K MEMORY EXPANDER ... 8K RAM memory expansion cartridge.

VIC 16K MEMORY EXPANDER ... 16K RAM memory expansion cartridge.

RS232 TERMINAL INTERFACE ... adapter cartridge for RS232 applications (connects to user port).

IEEE-488 INTERFACE CARTRIDGE ... for IEEE applications & PET/CBM accessories.

GAME JOYSTICK ... for playing Commodore games on cartridge or tape.

TWO PLAYER PADDLES ... for games and other programs.

LIGHTPEN ... for screen-touch programs such as computerized drawing.

PROGRAMMING AIDS

PROGRAMMER'S AID CARTRIDGE ... more than 20 BASIC program editing commands.

VIC SUPEREXPANDER CARTRIDGE ... graphics plotting, music, 3K expansion all on one cartridge.

VICMON ... Machine Language Monitor for writing/editing machine code programs.

PROGRAMMABLE CHARACTER/GAMEGRAPHICS EDITOR ... tape program lets you create your own VIC characters, symbols, alphabets.

TEACH YOURSELF PROGRAMMING SERIES ... self-teaching books and tapes for VIC owners who want to learn more about programming. Begins with INTRODUCTION TO PROGRAMMING, Part I.

INDEX

INDEX

A

Abbreviating sound commands, 96
Abbreviations, BASIC commands,
79, 263
ABS function, 41
ASC function, 41
Accumulator, 126, 140
Addition, 62
Addressing, VIC, 113
Adventure games, ix
AND operator, 65, 68
Applications, ix
Arithmetic formulas, 62, 275
Arithmetic operators, 62
Arrays, 60, 81
ASC function, 41
ASCII & CHR\$ Codes, 272
Asterisk (multiplication), 64
ATN function, 42
Auxiliary color, 93, 217

B

BASIC, 1
abbreviations, 79, 263
commands, 5
keyword codes, 121
locations, 116-120, 118-119
operators, 62, 68
statements, 14
variables, 58, 80
Beginning machine code, 132, 168
Bit mapping, 88
Bit patterns, 93
Boolean operators, 62
Boolean truth table, 66
Border color, 93
Buffer, 77
Bus
address, 109
control, 111
data, 111
Byte, 131

C

Calculator mode, 75
Character generator ROM, 82
Character memory, 82
Character size, 215
Chess (Sargon II), x
CHR\$ function, 40, 42
CHR\$ codes, 272
Chips
6502 chip, 113
VIC chip, 113
CLR statement, 14
CLR/HOME key, 73
Clock, 179, 184-5, 204
CLOSE statement, 35

CMD statement, 35

Color

auxiliary, 217
border, 217
keys, 29
memory map,
register, 93
screen and border,
Columns, video, 214
Commands, BASIC, 5
Commodore key, 73
Communication, x
Concatenation, 58, 62, 69
Connecting the VIC (see owners
guide)
CONT command, 5
Control bus, 111
Converting PET to VIC, 278
COSine function, 43
CRSR keys, 28, 73, 74
Crunching BASIC programs, 79
CTRL key, 29

D

DATA statement, 15, 81, 86
DELeTe key, 73
Deriving math functions, 275
Device addressing, 38
Device number addressing, 38
DIMension statement, 17, 61
Direct mode, 75
Disk, 8

E

Editing programs, 74
Eliminating spaces, 81
END statement, 18
Error messages, 276
Expansion port, 241, 244
Expansion RAM/ROM, 118
EXponent function, 43
Exponentiation, 63

F

Fetch cycle, 109
Filenames, 70
Floating point variables, 54, 59
FOR statement, 19
FOR . . . NEXT loop, 19
FRE function, 43, 85
Frequency modulation, 101
Functional block diagram, 110
Functions, BASIC, 40
Function keys, 78

G

Game controls, 246
Game port, 246
GET statement, 20, 77
GET# statement, 36
GOSUB statement, 20, 81, 133

GOTO statement, 22
Graphics, 82
 character memory, 82
 programmable characters, 82
 high resolution, 87
Greater than symbol, 64

H

Hexadecimal notation, 128, 131,
 170, 178
High resolution, 87
Home position, 73
Horizontal screen origin, 213

I

IEEE-488 Interface, 244
IF . . . THEN statement, 22
Immediate mode, 75
Indexed indirect addressing, 135
Indexing, 34
Indirect indexed addressing, 134
Input buffer, 77
INPUT statement, 24
INPUT# statement, 36
INSert key, 30, 73
Instruction set (6502), 140
INTEger function, 44
Integer variables, 54, 57
Interface mode, 213
Interpreter, BASIC, 119, 125
I/O guide, 227
I/O ports, addressing, 113, 184
I/O registers, 218
I/O statements, 35
I/O status, 49
IRQ, 243

J

Joystick, 246
Jump table, 138

K

Keyboard buffer, 77, 180
Keywords, 120-121
KERNAL, 114, 116, 125, 138, 182,
 251, 259
 power-up activities, 211
 user callable routines, 184

L

LDA (load accumulator), 130
LEFT\$ function, 44
LENGth function, 45
LET statement, 25
Light pen, xii, 215, 250
Line numbers, 79, 120
LIST command, 6
LOAD command, 7
Loan/Mortgage Calculation, xii
LOGarithm function, 45
Logical operators, 68

M

Machine language programming,
 107, 123
Machine code
Memory expansion, 124, 244
Memory map, 124, 170
Microprocessor (6502), 109
MID\$ function, 45
Minus sign, 63
Mixing sound & graphics, 105
Multicolor mode, 92
Multiple speakers, 100
Multiple statements on a line, 75,
 80
Music, 95, 96, 232
Musical note table, 266
Music, frequency modulation, 101
Music programming techniques, 98

N

NEW command, 9
NEXT statement, 26
NOT operator, 68
Number bases, 128
Numbers, 54

O

Octave comparison chart, 99
Octaves, 95
ON statement, 26
Operators, 62
OR operator, 68

P

Paddles, 216, 246
Parantheses (in formulas), 64
PEEK function, 46
Piano program, 103
Pin configuration, 213, 241
Pinouts for I/O devices, 280
POKE statement, 27
POS function, 46
PRINT statement, 28
PRINT#, 39
Printer, 236
Program counter, 126
Programmable characters, 82, 237
Programming music, 98
Programming tips, 71
Program mode, 75
Programs, 76
 editing, 73
 line numbering, 78

Q

Quote mode, 29

R

RAM memory, 85, 109, 111, 115
RAM starting locations, 85, 118
Raster value, 215

READ statement, 31, 81
 Register, 126
 Relational operators, 62, 64
 REMark statement, 31, 80
 Reserved Words, 60, 120
 Reset, 243, 4
 RESTORE statement, 32
 Return key, 73
 RETURN statement, 32
 Reversed characters, 29, 85, 217, 239
 RIGHT\$ function, 47
 RND function, 47
 ROM, 109, 114, 115
 Rounding numbers, 54
 Rows, video, 215
 RS-232 interface, 251
 RUN command, 10
 RUN/STOP RESTORE, 4, 25

S

SAVE command, 10
 Schematic (inside back cover)
 Scientific notation, 56
 Screen & border colors, 265
 Screen display codes, 267
 Screen editing, 73
 Screen formatting, 201
 Screen memory location, 215, 270
 Screen RAM, 115
 Serial bus, 234
 SGN function, 48
 Shift register, 221
 Shortening programs, 79
 SINE function, 48
 Sound commands, 96
 Sound, programming, 95, 216
 Space, 74, 81
 SPC function, 48, 81
 Speakers, 95
 SQR function, 49, 91
 ST numeric value, 50
 Stack, 133, 141
 Stack pointer, 127, 134
 Start of text, 119; of memory, 124
 Statements, BASIC, 14
 Status function, 49
 Status register, 126
 STOP command, 33, 4
 STR\$ function, 50
 String comparisons, 70
 String operations, 57, 70
 String variables, 57

Subroutines, 138-9
 Subtraction, 62
 Super expander, 94
 SYS statement, 33
 System clock, 204
 System overview, 109

T

TAB function, 51, 81, 121
 Talking VIC, xiv
 TAN function, 51
 TI variable, 52
 TI\$ variable, 52, 77, 179
 Time, setting VIC clock, 52
 Timer, 220
 Top of memory, 119
 True/false testing, 65, 68

U

Upper/lower case, 115
 USR function, 52
 User-defined function, 52
 Useful memory locations, 178
 User port, 229
 User program, memory location, 119

V

VALue function, 53
 Variables, 58, 80
 Variables, extended names, 60
 VERIFY command, 12
 Versatile interface devices, 109, 111, 218
 Vertical screen origin, 214
 VICMON, 127, 135, 137
 VICTIPS, 85, 96, 103, 105
 Video interface chip, 116, 212
 Volume, 95

W

WAIT statement, 34
 Warm start, 4
 White noise generator, 104
 Writing machine code, 132

X

X Index register, 126

Y

Y Index register, 126

Z

Zero page, 133

THE MICROCOMPUTER MAGAZINE

commodore

The Commodore Magazine provides a vehicle for sharing the latest product information on Commodore systems, programming techniques, hardware interfacing, and applications for the CBM, PET, SuperPET and VIC systems.

Each issue contains features of interest to anyone that owns, or is thinking about, purchasing Commodore equipment:

- Application articles examine how users from various backgrounds use Commodore computers in business, education, and the home.
- Columns by leading experts explain ways to get the most out of your computer in clear, concise language.
- The latest news on user clubs, a question/answer hotline column, and reviews of the newest books and software round out the most complete magazine devoted exclusively to Commodore computers.

Readers are also encouraged to submit articles for publication and share their experiences using Commodore computers.

Commodore Magazine is published 6 times a year by Commodore Business Machines, Inc. The subscription fee is \$15.00 for six issues within the United States and its possessions and \$25.00 for Canada and Mexico.

Commodore Magazine

Subscription Form

Name: _____

Address: _____

City: _____ State: _____ Zip: _____

Renewal Subscription _____ New Subscription _____

Equipment Use: ☐ Personal ☐ Education ☐ Business

My subscription began with issue number _____

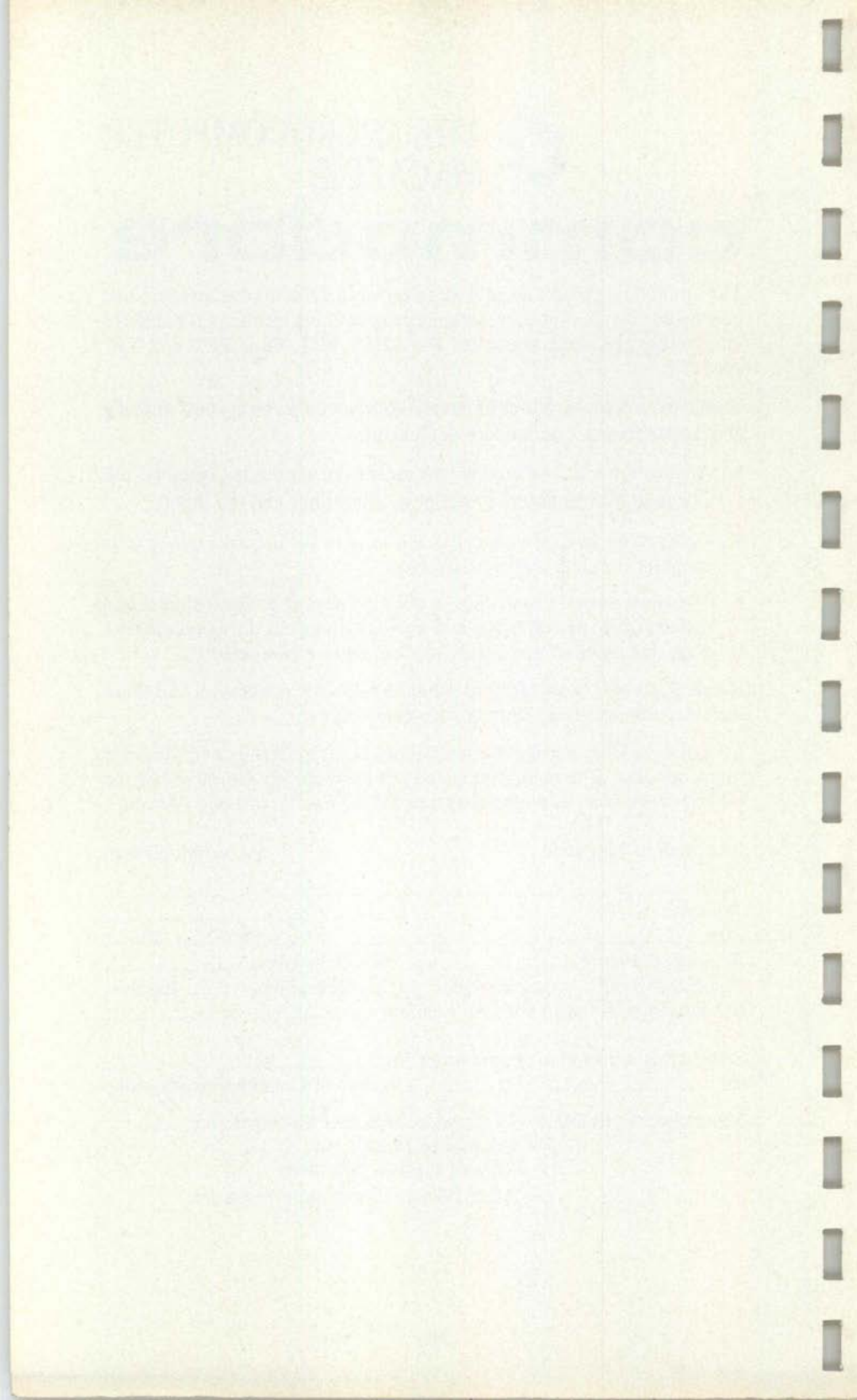
Enclosed is a check or money order for \$ _____
for _____ issues of Commodore Magazine

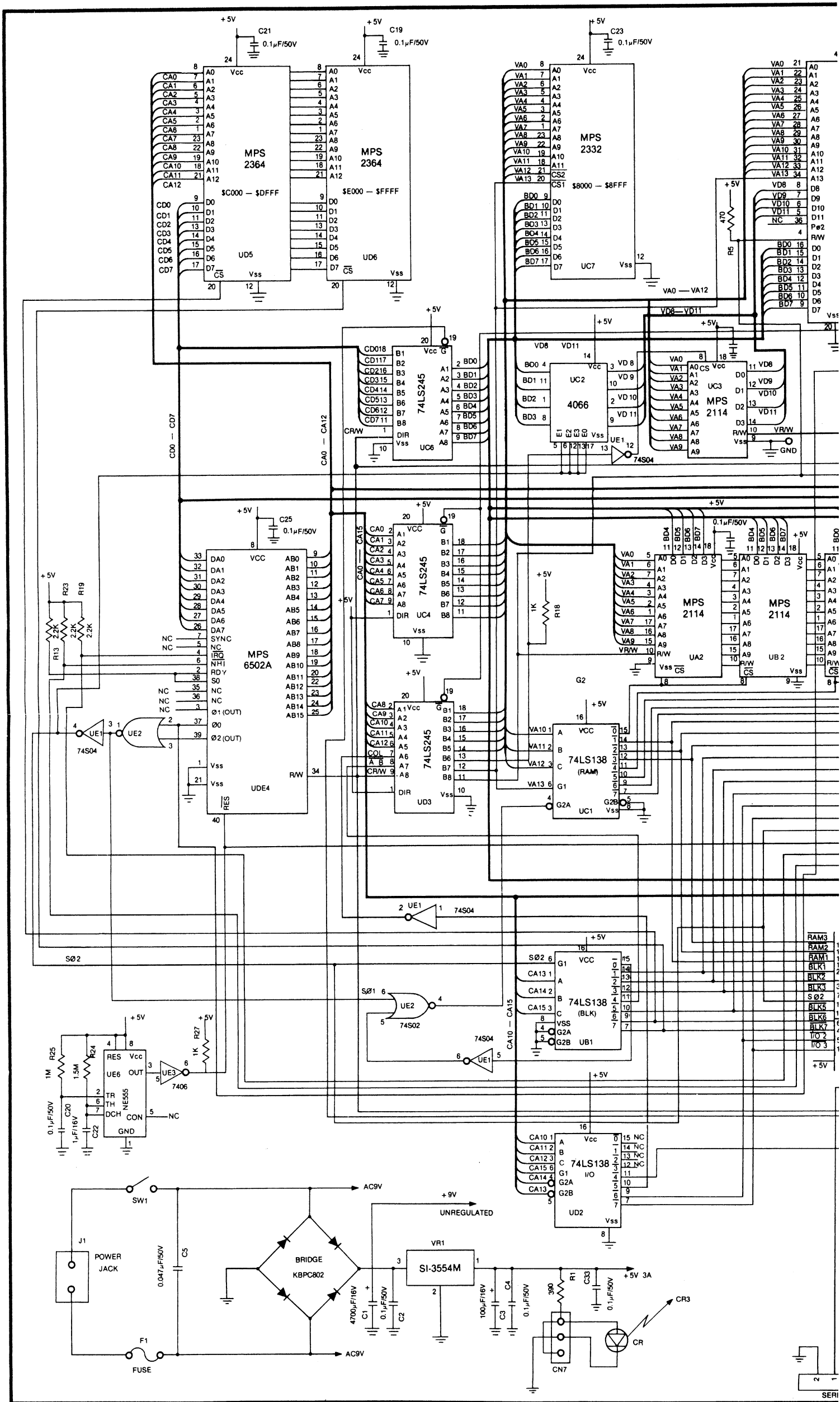
Make checks payable to Commodore Business Machines, Inc.

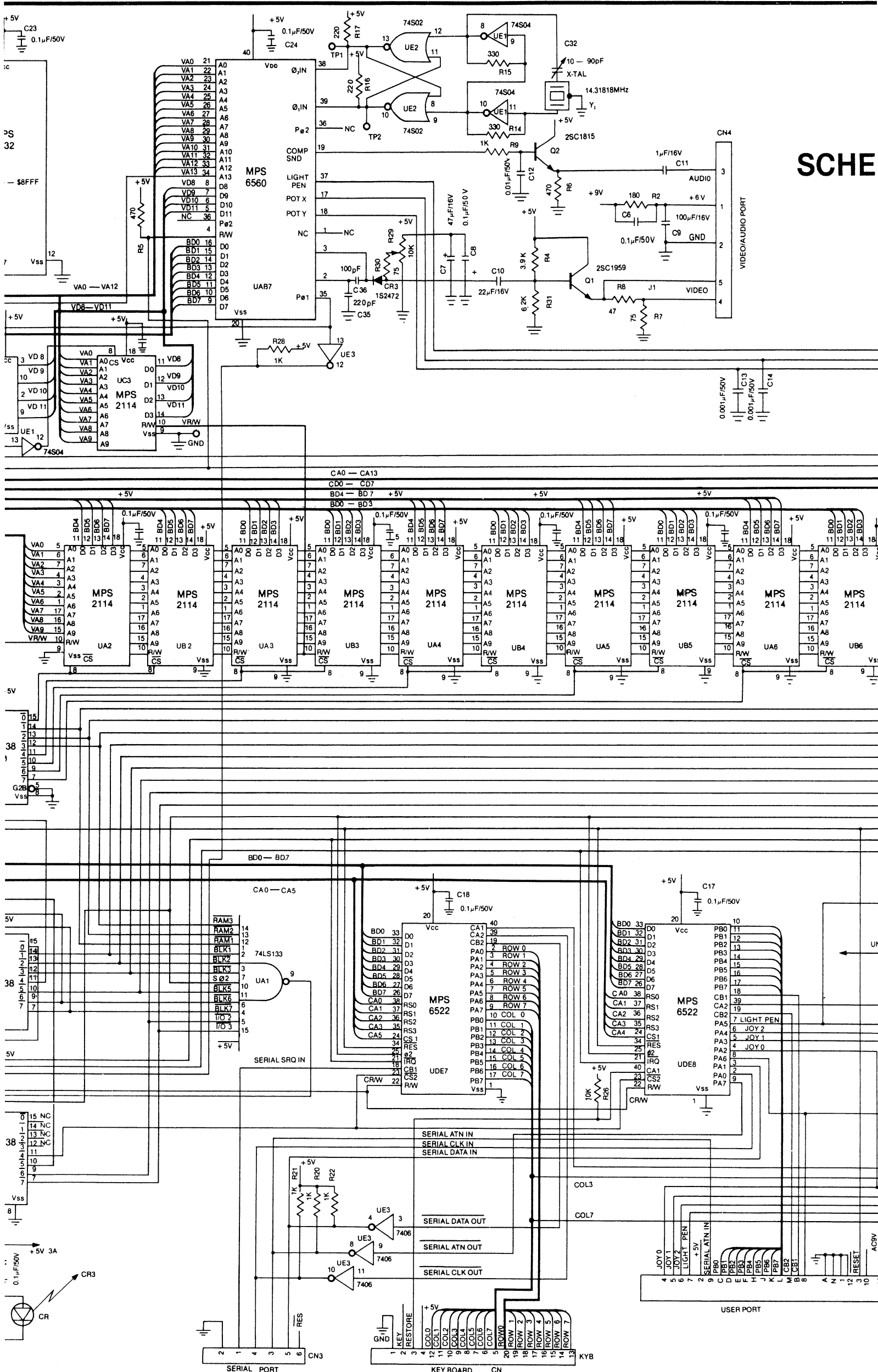
681 Moore Road

King of Prussia, PA 19406

Attn: Editor, Commodore Magazine







SCHE



ABOUT THE VIC 20 PROGRAMMERS REFERENCE GUIDE...

This easy-to-use manual gives you a ready source of information on VIC 20 software and hardware, with detailed explanations of each topic and "friendly" tips throughout to help you use your VIC 20 to best advantage.

The VIC PROGRAMMERS REFERENCE GUIDE is actually four guides in one. It includes (1) a BASIC VOCABULARY GUIDE which explains the complete VIC BASIC language instruction set along with (2) a PROGRAMMING TIPS GUIDE with suggestions on how to improve your programming, (3) a MACHINE LANGUAGE PROGRAMMING GUIDE to help you talk to the VIC in its "own" binary/hexadecimal language, and (4) a special section on INPUT/OUTPUT OPERATIONS giving you the information needed to connect your VIC to special peripherals like RS232 devices, lightpens and others.

This guide was compiled from information provided by Commodore programming staffs working in more than half a dozen countries worldwide.

Whether you're a first-time computerist or an expert programmer, you'll find a wide variety of programming aids available through your Commodore dealer. In addition to books and manuals, Commodore provides several special programming cartridges and the TEACH YOURSELF PROGRAMMING (tm) instruction series.



DISTRIBUTED BY

Howard W. Sams & Co., Inc.

4300 W. 62nd Street, Indianapolis, Indiana 46268 USA

\$16.95/21948

ISBN: 0-672-21948-4